VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Interprocedurální analýza vztahů mezi hodnotami proměnných programu v modulární aritmetice

Interprocedural Analysis of Relations on Values of Program Variables in Modular Arithmetic

2014

Bc. Michala Čandová

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Michala Čandová**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Interprocedurální analýza vztahů mezi hodnotami proměnných programu v modulární aritmetice
Interprocedural Analysis of Relations on Values of Program Variables in Modular Arithmetic

Zásady pro vypracování:

Cílem práce je implementovat algoritmus popsaný v článku Müller-Olm, Seidl (2007). Tento algoritmus slouží k automatickému nalezení afinních relací v modulární aritmetice modulo 2^k (pro nějaké fixní k, např. k = 32), které platí mezi hodnotami proměnných analyzovaného programu v jednotlivých bodech tohoto programu pro jeho všechny možné běhy. Afinní relace jsou relace typu a_1 * x_1 + a_2 * x_2 + ... + a_n * x_n = b, kde a_1, a_2, ..., a_n jsou číselné konstanty a x_1, x_2, ..., x_n jsou hodnoty proměnných programu v daném bodě.
Jedná se interprocedurální analýzu, kde se analyzovaný program může skládat z libovolného počtu procedur, které se mohou navzájem (rekurzivně) volat. Analýza pak bere v úvahu vzájemné volání procedur a kontext jednotlivých volání.

1. Nastudujte příslušnou problematiku.
2. Implementujte algoritmus popsaný v článku Müller-Olm, Seidl (2007).
3. Implementovaný algoritmus otestujte na vhodně zvolených testovacích programech.

Seznam doporučené odborné literatury:

M. Müller-Olm, H. Seidl: Analysis of Modular Arithmetic, ACM Transactions on Programming Languages and Systems (TOPLAS) - Special Issue ESOP'05 TOPLAS, Volume 29, Issue 5, pp. 29:1-29.27, ACM, 2007.
Další literatura podle pokynů vedoucího práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Zdeněk Sawa, Ph.D.**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

5. 5. 2014 .....

.....

Ráda bych tímto poděkovala vedoucímu mé diplomové práce, doc. Ing. Zdeňku Sawovi, Ph.D., za jeho odborné vedení, cenné rady a připomínky, které mi po celou dobu poskytoval.

Abstrakt

Cílem této diplomové práce je implementace algoritmů navržených v odborném článku Analysis of Modular Arithmetic (Müller-Olm, Seidl), sloužících k výpočtu vztahů mezi hodnotami proměnných. Algoritmy jsou implementovány jako součást aplikace, která provádí interprocedurální analýzu programu napsaného v jazyce navrženém pro potřeby této práce. Výstupem aplikace je soubor obsahující původní program doplněný o vypočtené vztahy mezi hodnotami proměnných programu. Kromě implementace navržených algoritmů se práce zabývá také formální kontrolou správnosti načteného programu a jeho úpravou pro následné zpracování navrženými algoritmy.

Klíčová slova

diplomová práce, interprocedurální analýza, intraprocedurální analýza, statická analýza, modulární aritmetika, modulo, množina generátorů, afinní relace, lineární rovnice

Abstract

The aim of this thesis is the implementation of algorithms proposed in article Analysis of Modular Arithmetic (Müller-Olm, Seidl), used to compute relations between values of variables. The algorithms are implemented as a part of the application performing an interprocedural analysis of program written in special language designed for needs of this thesis. The output of this application is a file with original program and calculated relations on values of program variables. Except for implementation of proposed algorithms the thesis also deals with formal proof of correctness of the loaded program and its adjustment for designed algorithms processing.

Keywords

thesis, interprocedural analysis, intraprocedural analysis, static analysis, modular arithmetic, modulo, set of generators, affine relations, linear equations

Seznam použitých symbolů a zkratek

CFG - graf toku řízení (z angl. control flow graph)
DKA - deterministický konečný automat
AST - abstraktní syntaktický strom

Obsah

1 Úvod

Cílem této diplomové práce je implementace algoritmů navržených Prof. Dr. Markusem Müller-Olmem a Prof. Dr. Helmutem Seidlem v odborném článku *Analysis of Modular Arithmetic*. Tyto algoritmy byly implementovány jako součást aplikace, která zpracovává program, načtený na vstupu této aplikace. Načtený program musí být napsán v jednoduchém programovacím jazyce, který byl navržen speciálně pro potřeby této práce. Tento jazyk pracuje pouze s globálními proměnnými celočíselného typu. Umožňuje však použití neomezeného množství funkcí, které se mohou navzájem rekurzivně volat. Hlavním úkolem implementované aplikace je vypočtení lineárních vztahů mezi hodnotami proměnných pro každé místo v kódu načteného programu. Vypočtené lineární vztahy platí v modulární aritmetice, kde se veškeré operace provádějí v modulo 2^w, kde w je počet bitů. Stanovení rozsahu, v jakém mají výpočty probíhat (tedy stanovení počtu bitů), se provádí parametrem při spouštění aplikace. Výstupem aplikace je nový soubor obsahující původní program doplněný o výpis platných lineárních vztahů za každým příkazem v programu.

Aplikace je implementována v programovacím jazyku C#. Aplikace byla otestována na různých vstupních programech a v rámci testování byla sledována doba zpracování navržených algoritmů s cílem zjistit faktory, které mohou dobu zpracování algoritmů ovlivnit.

Důvody, které je vedly k návrhu algoritmů dle [2], spočívaly především v tom, že běžné programovací jazyky pracují v modulární aritmetice modulo mocnina 2 a tudíž mají nulové dělitele. To znamená, že některá pravidla běžně uplatňovaná pro výpočty např. v reálných číslech (obecně v tělesech) nelze přímo použít. Navržené algoritmy výpočty v modulární aritmetice zohledňují a díky tomu umožňují vyvození platných afinních vztahů mezi programovými proměnnými v každém bodu analyzovaného programu. Algoritmy zohledňují i podmínkové větvení programu, které nahrazují nedeterministickým větvením, díky čemuž detekují všechny platné afinní vztahy.

Jelikož navržené algoritmy patří do oblasti statické analýzy, věnuje se druhá kapitola této práce popisu a významu statické analýzy pro lepší pochopení později popsaných použitých postupů.

Třetí kapitola se zabývá předzpracováním analyzovaného programu, neboť navržené algoritmy pracují s programem zpracovaným do podoby grafu toku řízení (angl. Control Flow Graph, dále jen CFG). Bylo tedy třeba nejprve navrhnout gramatiku a pro ni vytvořit příslušné postupy pro úpravu programu, zapsaného v dané gramatice, do podoby CFG. Postupně tedy byly pro navrženou bezkontextovou gramatiku vytvořeny lexikální analýza, syntaktická analýza a sémantická analýza, které ověřují, že načtený program splňuje pravidla definovaná gramatikou. Výstupem syntaktické analýzy je abstraktní syntaktický strom (dále jen AST), který je použit při vytváření CFG jednotlivých funkcí, které jsou pak vstupem pro samotné algoritmy.

Čtvrtá kapitola se věnuje postupům souvisejícím s dalším zpracováním CFG již v souvislosti se zpracováním samotnými algoritmy. Jelikož algoritmy očekávají zpracování výpočtů jako násobení matic a vektorů, zabývá se tato kapitola postupy sestavení vektorů a matic. Dále vysvětluje pravidla modulární aritmetiky se zaměřením na speciální případ modulo 2^w a vysvětluje principy implementovaných algoritmů pro výpočet interprocedurální analýzy.

Pátá kapitola se zabývá návrhem a implementací aplikace pro zpracování analyzovaného programu. V rámci návrhu dochází k pomyslnému rozdělení aplikace na 2 části a to na část zajišťující předzpracování analyzovaného programu do podoby CFG a na část řešící samotnou interprocedurální analýzu. Dále kapitola obsahuje popis jednotlivých tříd, které jsou v aplikaci implementovány.

Poslední kapitola je věnována testování aplikace. Nejprve je na jednoduchých příkladech ověřeno, zda jsou výsledky analýzy v daných bodech programu opravdu správné a tedy zda analýza vrací správné výsledky a následně je zmapována rychlost výpočtu a její ovlivnění počtem proměnných.

2 Statická analýza

Statická analýza se zabývá studiem programového kódu, aniž by došlo ke spuštění programu. Používá se např. pro ověření korektnosti programu, optimalizace programového kódu, ověření, zda proměnné programu nabývají pouze takových hodnot, kterých nabývat mají, nebo pro ověření, zda program po optimalizaci algoritmu má stejné chování, jako původní.

Mezi důležité postupy při řešení některých typů úloh a při studiu vlastností různých algoritmů se často používá hledání invariantů, díky kterým lze odvodit řešení zadané úlohy (resp. dokázat, že daná úloha řešení nemá) nebo popsat některé vlastnosti daného algoritmu. Poněkud specifickým případem mohou být podmínky cyklů v jakémkoli programovém kódu. Obecně lze říci, že hledání invariantů může vést k zajímavým zjištěním o sledovaném programu. Cílem této práce je implementace algoritmu navrženého v článku [2], který umožní sledovat vztahy mezi programovými proměnnými v každém bodu programu pomocí afinních vztahů. Díky tomu umožní další studium chování daného algoritmu.

Pro analýzu zkoumaného programu jsou využity metody statické analýzy. Statická analýza slouží ke zkoumání vlastností programu bez jeho spuštění. Lze ji použít například pro optimalizaci kódu, odhalení nepoužitých proměnných nebo nalezení chyb v programu.

Základem pro zpracování statické analýzy je sestavení CFG, pro každou funkci zkoumaného programu. CFG je orientovaný graf, jehož vrcholy odpovídají jednotlivým bodům programu a hrany reprezentují jednotlivé příkazy [3]. Statická analýza pak pomocí definovaných algoritmů prochází vytvořený CFG. Podle toho, zda se analýza zabývá každou funkcí zkoumaného programu samostatně, nebo zda se zabývá i vazbami mezi jednotlivými funkcemi programu, se dělí na intraprocedurální a interprocedurální.

Intraprocedurální analýza vždy zkoumá vlastnosti pouze jedné funkce, pokud je v programu funkcí více, pracuje vždy s každou funkcí samostatně. Mezi problémy, kterými se intraprocedurální analýza zabývá, patří například živost proměnných, dostupnost výrazů, opakované vyhodnocování výrazů[1] nebo dosažitelnost definicí. Interprocedurální analýza se pak zabývá řešením těchto problémů v souvislosti všech funkcí, které zkoumaný program má. Zohledňuje tedy i volání funkcí a jeho vliv na zkoumané problémy.

Pro potřeby implementovaného algoritmu jsou využity především postupy týkající se monovariantní interprocedurální analýzy a algoritmu pevného bodu. Monovariantní interprocedurální analýza pracuje se všemi funkcemi tak, jak jsou reprezentovány ve zkoumaném programu, oproti polyvariantní analýze, která vytváří kopii funkce pro každé její volání. Tím umožňuje zachování přesnějších vztahů mezi proměnnými, které jsou v tomto případě analýzou sledovány. Monovariantní interprocedurální analýza prochází CFG funkce `main` (obecně první zpracovávané funkce zkoumaného programu) a v okamžiku, kdy na hraně CFG načte volání funkce, předá na vstupní bod volané funkce aktuální hodnoty proměnných z vrcholu CFG, ze kterého vychází hrana obsahující volání funkce. Do vrcholu CFG, do kterého hrana s voláním funkce vstupuje, pak jsou uloženy návratové hodnoty proměnných z funkce volané na hraně.

Aby bylo zajištěno zpracování všech vrcholů všech CFG a zároveň předání všech možných hodnot, které mohou být v průběhu průchodu CFG několikrát měněny (např. v různých cyklech nebo při iterativních voláních funkcí), je využit algoritmus pevného bodu, který umožňuje implementovat konečné algoritmy tak, aby byl výpočet konečný. Na počátku výpočtů jsou množiny hodnot uložené ve všech vrcholech všech CFG nastaveny na prázdné množiny a je

[1] Anglicky dle [3] „very busy expressions“

vytvořena fronta vrcholů ke zpracování, do které jsou vloženy všechny vrcholy všech CFG. Následně je odebrán první vrchol z fronty a zpracován. Zpracováním se rozumí načtení příkazu postupně z každé hrany, která z daného vrcholu vychází a provedení tohoto příkazu. Pokud provedením příkazu dojde ke změně hodnoty, která je uložena ve vstupním vrcholu zpracovávané hrany, bude vstupní uzel hrany vložen na konec fronty k dalšímu zpracování. Popsaný postup zajišťuje, že algoritmus snadno rozpozná, když se hodnoty ve vrcholech přestanou měnit, protože fronta vrcholů ke zpracování zůstane prázdná. To zároveň znamená, že byly zpracovány všechny vrcholy a obsahují všechny možné změny, které je mohou ovlivnit.

3 Zpracování vstupního programu

Aby bylo možné na vstupním programu provést interprocedurální analýzu, je třeba nejprve převést kód tohoto programu z formátu prostého textu do strukturované podoby CFG. Pro převod do CFG je využit syntaktický strom, který je výstupem syntaktické analýzy. Je tedy nutné nejprve definovat gramatiku pro vstupní program a při načítání programu pak provést lexikální, syntaktickou a sémantickou analýzu. Lexikální, syntaktická i sémantická analýza jsou obecně součástí překladače ke každému programovacímu jazyku. S ohledem na skutečnost, že pro účely této práce byl navržen specifický programovací jazyk, byl také implementován příslušný překladač.

3.1 Gramatika vstupního programu

S ohledem skutečnost, že algoritmus pracuje pouze s přirovnáními obsahujícími lineární výrazy na pravé straně a všechny ostatní označuje speciálním symbolem a obecně předpokládá, že v daném místě mohou proměnné nabývat všech hodnot, je tomu uzpůsobena i bezkontextová gramatika pro vstupní program. Ukázka funkce v definované gramatice je na Obrázku 1.

```
function calculation
{
    if(a > b){
       c = 4 + 3a - 2b;
       b++;
    }
    else{
       c = 3 * (b - a);
       a++;
    }
    a = b + 3c;
}
```

Obrázek 1 - Ukázka funkce v definované gramatice

Navržená gramatika předpokládá použití pouze celočíselných proměnných v decimálním nebo hexadecimálním formátu bez předchozího rozlišení datového typu, rozlišuje názvy proměnných a funkcí pomocí klíčových slov `var` a `function` a umožňuje definici pouze globálních proměnných. Dále gramatika umožňuje použití libovolného množství funkcí a vyžaduje existenci funkce `main`. Funkce se mezi sebou mohou libovolně (i rekurzivně) volat. Dále gramatika umožňuje použití příkazů pro řízení toku programu `if-else`, `while`, `for` a `goto` a příkaz `return`. Povolený formát přirovnání je podřízen požadavkům algoritmu, a proto připouští pouze tvary přirovnání s lineárními výrazy na pravé straně. Popis všech pravidel bezkontextové gramatiky je uveden v příloze č. I.

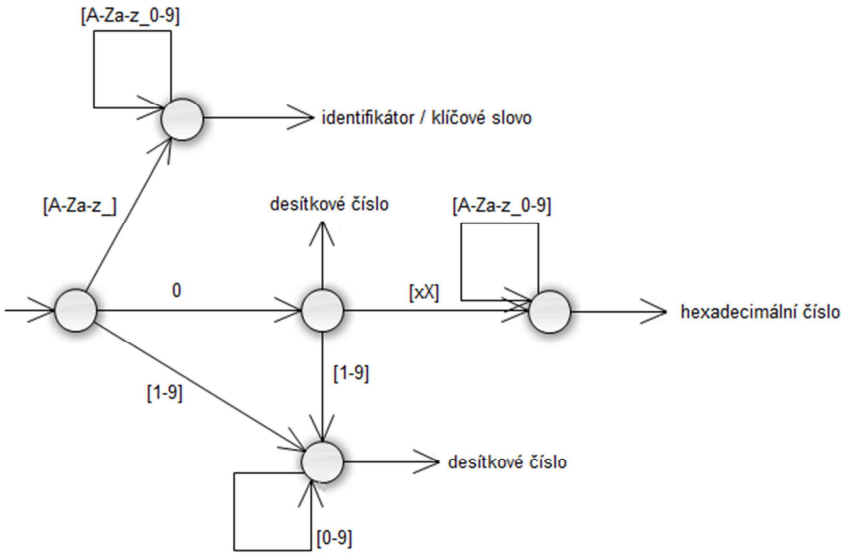
3.2 Lexikální analýza

Úkolem lexikální analýzy je načtení zdrojového textu a vrácení jednotlivých slov jako celků, tzv. tokenů. Lexikální analýza vrací nejen jednotlivá slova zdrojového textu, ale zároveň

rozpoznává, zda se jedná o klíčová slova, identifikátory, čísla nebo operátory. Každý token tedy vždy obsahuje načtený řetězec znaků, které tvoří slovo (tzv. lexém), a zároveň i informaci o typu lexému (tj. zda se jedná o klíčové slovo, identifikátor, apod.).

Nástroj provádějící lexikální analýzu, tzv. lexikální analyzátor, načítá postupně jednotlivé znaky zdrojového textu a podle předdefinovaných pravidel, vycházejících z bezkontextové gramatiky, rozpoznává a identifikuje jednotlivé tokeny. Postup zpracování zdrojového textu začíná načtením prvního znaku, podle kterého analyzátor zkusí rozhodnout, o jaký typ tokenu se jedná. Například u závorek nebo oddělovačů může rozhodnout ihned, jelikož jsou výhradně jednoznakové. V případě operátorů lze ve většině případů také rozhodnout podle prvního znaku, případně se analyzátor podívá na další následující znak a rozhodne podle něj. V případě, kdy je jako první znak načteno písmeno nebo číslo, analyzátor rozpozná slovo, resp. číslo, a přečte všechny znaky až do konce daného slova. Pokud je podle prvního znaku rozpoznáno slovo, kontroluje se po přečtení celého slova, zda se nejedná o klíčové slovo, v případě čísla je třeba zkontrolovat formát čísla.

Konec lexému je zpravidla rozpoznán podle načtení prázdného znaku, může však být načten i znak následujícího lexému (oddělovače, operátoru nebo závorky) nebo má aktuálně čtený a rozpoznaný lexém pevnou délku (operátory, závorky, oddělovače). Jakmile je přečten celý lexém a určen jeho typ, jsou tyto informace uloženy do tokenu. Lexikální analyzátor při své činnosti odstraňuje všechny prázdné znaky a komentáře.



Obrázek 2 - Částečný DKA lexikální analýzy

Každý lexikální analyzátor je v podstatě deterministickým konečným automatem (dále jen DKA), který podle definovaných pravidel rozhoduje o tom, jaké slovo bylo načteno. Při jeho běhu může nastat situace, kdy je načten znak, pro který v daném stavu neexistuje cesta. V takovém případě se lexikální analýza zastaví a vrátí chybu, která byla identifikována. Zjednodušený deterministický konečný automat, zachycující rozpoznávání čísel a identifikátorů (resp. klíčových slov) je zobrazen na Obrázku 2. Ze zobrazeného automatu byly vypuštěny větve týkající se přímého přechodu do koncových stavů při načtení závorek, operátorů a oddělovačů.

Na DKA z Obrázku 2 lze vidět, že v případě čtení písmen, je výstupem buď identifikátor, nebo klíčové slovo. V případě, že analyzátor skončí v tomto stavu, kontroluje v seznamu klíčových slov, zda v něm existuje slovo shodné s přečteným lexémem. Pokud ano, je lexém do tokenu uložen jako typ klíčové slovo, pokud ne, jedná se o identifikátor.

3.3 Syntaktická analýza

Cílem syntaktické analýzy je rozpoznat význam postupně čtených tokenů a na základě znalosti formální gramatiky zdrojového textu sestavit syntaktický strom, který zachycuje význam jednotlivých tokenů v rámci celku.

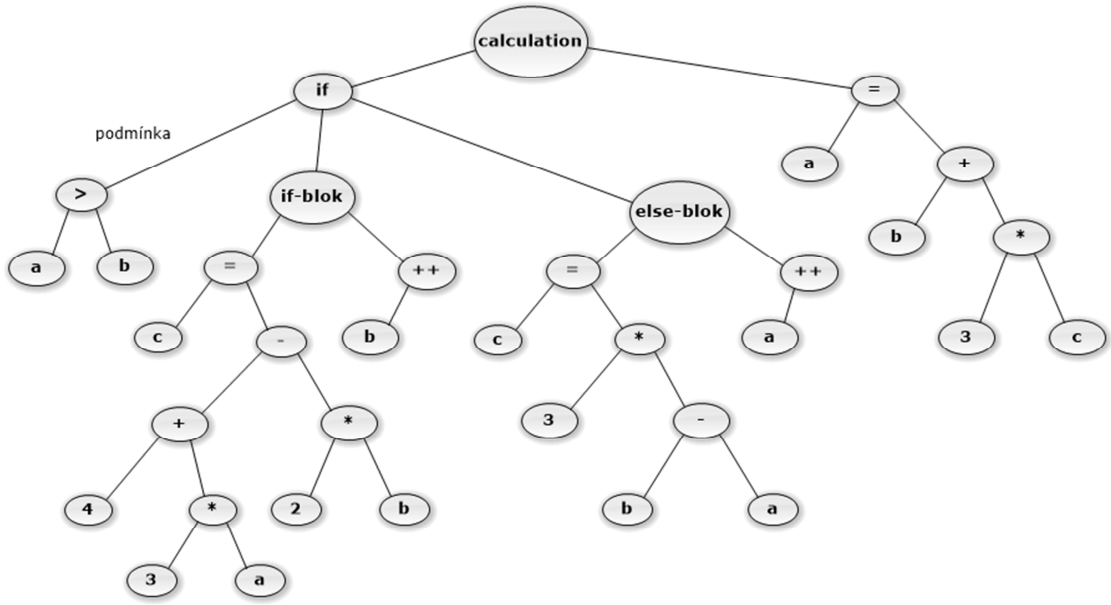
Syntaktický analyzátor, který syntaktickou analýzu provádí, rozpoznává funkce a proměnné, sestavuje příkazy a funkce, kontroluje správné použití závorek (stejný počet levých i pravých závorek stejného typu) a případné další požadavky na syntaxi definované formální gramatikou.

Podle způsobu práce syntaktického analyzátoru se rozlišují dva základní typy syntaktické analýzy – shora dolů a zdola nahoru. Syntaktická analýza zdola nahoru nejprve identifikuje terminální symboly a následně z nich sestavuje neterminální symboly, sestavuje tedy derivační strom od listů směrem ke kořenu (proto označení zdola nahoru). Oproti tomu syntaktická analýza shora dolů se snaží sestavit derivační strom od kořene k listům. Při implementaci syntaktické analýzy pro potřeby této práce byla využita analýza shora dolů, a proto se další text již týká pouze tohoto typu.

Pro příklad mějme jednoduchou gramatiku $S \rightarrow S + S \mid a \mid b \mid 1$ a řetězec $a + b + 1$. Při použití syntaktické analýzy shora dolů, může analýza vypadat takto:

1. S (celý řetězec je označen neterminálem S)
2. $S + S$ (neterminál S byl rozložen na součet neterminálů $S + S$)
3. $S + S + S$ (levý neterminál S byl rozložen na součet neterminálů $S + S$)
4. $a + S + S$ (levý neterminál S byl nahrazen terminálem a)
5. $a + b + S$ (levý neterminál S byl nahrazen terminálem b)
6. $a + b + 1$ (neterminál S byl nahrazen terminálem 1)

Výstupem syntaktické analýzy je AST (někdy také jen syntaktický strom). AST zachycuje strukturu vstupního řetězce ve stromové struktuře, kde uzly reprezentují funkce, jejich části a operátory a listy reprezentují operandy. AST nezachycuje strukturu vstupního řetězce doslovně, tak jak by ji zachytil derivační strom, ale v abstraktní zjednodušené podobě, která umožní např. zachycení konstrukce `if-then-else` v jediném uzlu se dvěma větvemi. Platí, že každý podstrom je samostatnou logickou jednotkou [4]. Obrázek 3 zachycuje ukázku AST pro funkci `calculation` z Obrázku 1.



Obrázek 3 - Abstraktní syntaktický strom

3.4 Sémantická analýza

Sémantická analýza navazuje na výstup syntaktické analýzy a kontroluje, zda použité proměnné a funkce byly řádně deklarovány a zda jsou v daném kontextu správně používány. Obecně platí, že sémantická analýza kontroluje i použití proměnných ve vazbě na jejich datový typ, případně provádí konverzi datového typu (pokud je to možné), a v případě funkcí rozlišuje funkce podle jejich parametrů (pokud daný programovací jazyk podporuje přetěžování funkcí).

Pro potřeby této práce je dostačující využití sémantické analýzy pouze v omezené míře, neboť programovací jazyk, navržený pro vstupní program, nepodporuje lokální proměnné ani vnořené funkce. Realizace sémantické analýzy je tedy nutná pouze v rozsahu kontroly deklarace funkcí, které jsou volány. To znamená, že sémantická analýza pouze prochází AST vytvořený syntaktickou analýzou a na všech místech, kde zjistí volání funkce, ověřuje, zda byla daná funkce deklarovaná. Seznam deklarovaných funkcí je vytvořen již syntaktickou analýzou. S ohledem na skutečnost, že lokální proměnné podporovány nejsou, je možné deklaraci použitých proměnných ověřovat již v průběhu syntaktické analýzy.

Provedením sémantické analýzy nedochází ke změně AST vytvořeného syntaktickou analýzou. Sémantická analýza pouze provádí kontrolu AST s ohledem na deklarace a volání funkcí.

3.5 Vytvoření CFG

Pro vytvoření CFG je třeba převést vytvořený AST na posloupnost příkazů, obsahující pouze příkazy `if` a `goto-label`. Pokud tedy gramatika vstupního programu umožňuje použití i dalších příkazů, jako je `while`, `for` nebo `if-else`, je třeba zavést pravidla, podle kterých budou tyto příkazy převedeny do požadovaného formátu. Pro zavedení potřebných pravidel je nutné si uvědomit, jak se jednotlivé příkazy chovají.

V případě příkazu `while` je vždy vyhodnocena podmínka a pokud je splněna, provede se blok příkazů patřící k příkazu `while`. Po provedení posledního příkazu bloku se běh programu vrátí na začátek bloku a opět vyhodnotí podmínku. Dokud je podmínka splněna, provádí se opakovaně příkazy v bloku `while`, v okamžiku, kdy podmínka splněna není, přeskočí běh programu na první příkaz za tímto blokem a pokračuje následujícími příkazy. Při převodu na posloupnost příkazů řízenou pouze příkazy `if` a `goto-label` se tedy jeví jako nejvhodnější řešení negování podmínky spuštění bloku `while` a pokud je splněna tato negovaná podmínka, pak běh programu přeskočí na první příkaz, který již nepatří do bloku `while`. Pokud negovaná podmínka splněna není, pak se provedou všechny příkazy z bloku `while` a na jejich konci běh programu skočí zpět na vyhodnocení negované podmínky. Příklad převodu bloku `while` je uveden v Tabulce 1.

Blok příkazů	**Převod na posloupnost příkazů `if` a `goto-label`**

`1: while (i < 10){`
`2:  a = b + i;`
`3:  i++;`
`  }`
`4: ...`
	`1: if !(i < 10) goto 5`
	`2: a = b + i`
	`3: i++`
	`4: goto 1`
	`5: ...`

Tabulka 1 - Převod bloku příkazů `while`

Pokud je při běhu programu přečten příkaz `if`, vyhodnocuje se podmínka a pokud je splněna, vstupuje běh programu do bloku příkazu `if`. V případě, že podmínka splněna není, ověří se, zda existuje blok `else`. Pokud existuje, pokračuje běh programu provedením příkazů v bloku `else` a po jeho skončení pokračuje dalšími příkazy za tímto blokem. Pokud blok `else` neexistuje, skočí běh programu na první příkaz za blokem `if` a pokračuje postupným prováděním dalších příkazů. Pro převod na posloupnost příkazů je tedy jako první opět nejprve vyhodnocena negovaná podmínka. Pokud není splněna a neexistuje blok `else`, pokračuje se dále čtením dalších příkazů. Pokud existuje blok `else`, je třeba za poslední příkaz bloku `if` přidat příkaz `goto`, který zajistí, že budou přeskočeny příkazy z bloku `else` a bude se pokračovat až dalšími příkazy.

Blok příkazů	**Převod na posloupnost příkazů `if` a `goto-label`**

`1: if (i < 10){`
`2:  a = b + i;`
`  }`
`3: ...`
	`1: if !(i < 10) goto 3`
	`2: a = b + i`
	`3: ...`

`1: if (i < 10){`
`2:  a = b + i;`
`  }`
`3: else {`
`4:  a = b - i;`
`  }`
`5: ...`
	`1: if !(i < 10) goto 4`
	`2: a = b + i`
	`3: goto 5`
	`4: a = b - i`
	`5: ...`

Tabulka 2 - Převod bloku příkazů `if` a `if-else`

Pokud je negovaná podmínka vyhodnocena jako pravdivá, pak je třeba přejít na první příkaz bloku `else`, pokud existuje, nebo na první příkaz za blokem `if`, pokud blok `else` neexistuje. Příklady převodu bloku `if` a `if-else` jsou uvedeny v Tabulce 2.

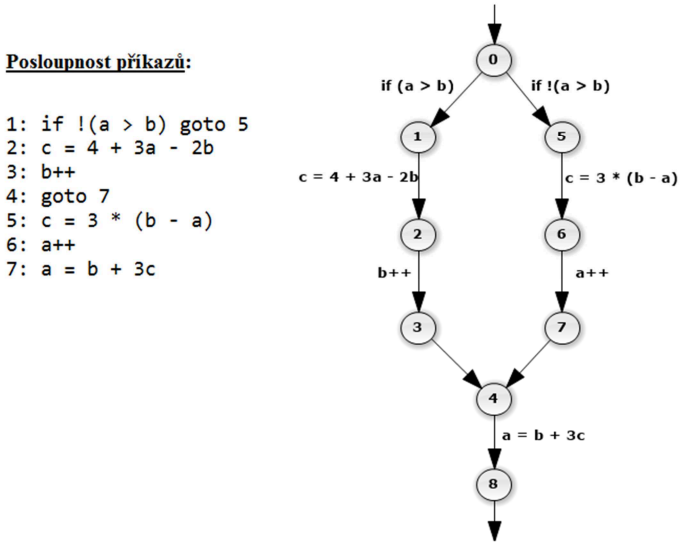
Vyhodnocení příkazu `for` začíná přečtením příkazu pro nastavení vstupní hodnoty, následně dochází k vyhodnocení podmínky, a pokud je splněna, vstupuje běh programu do bloku příkazů náležejícího k čtenému příkazu `for`. Po přečtení posledního příkazu daného bloku je proveden příkaz uvedený jako poslední v deklaraci příkazu `for` a znovu se vyhodnotí podmínka. Jakmile není splněna podmínka, pokračuje program prováděním prvního příkazu za blokem `for`. Při převodu na posloupnost příkazů je tedy třeba jako první příkaz uvést příkaz nastavující vstupní hodnotu. Teprve pak je vyhodnocena negovaná podmínka, a pokud je splněna, přeskočí běh programu až na první příkaz za blokem příkazů náležejících do bloku `for`. Pokud negovaná podmínka splněna není, pokračuje program postupným přidáváním všech příkazů v bloku `for`. Po jejich přidání je třeba připojit jako další příkaz poslední příkaz z deklarace `for` a skok na opakované vyhodnocení podmínky. Příklad popsaného převodu je uveden v Tabulce 3.

Blok příkazů	**Převod na posloupnost příkazů `if` a `goto-label`**

`1: for (i = 0; i < 10; i++){`
`2:  a = b + i;`
`  }`
`3: ...`
	`1: i = 0`
	`2: if !(i < 10) goto 6`
	`3: a = b + i`
	`4: i++`
	`5: goto 2`
	`6: ...`

Tabulka 3 - Převod bloku příkazů `for`

Následným čtením vytvořené posloupnosti příkazů lze vytvořit CFG, který bude mít v hranách uloženy jednotlivé příkazy a do uzlů bude ukládat hodnoty vypočtené pomocí dále popsaných postupů. Ukázka vytvoření posloupnosti příkazů pro funkci `calculation` z Obrázku 1 a k ní náležející CFG jsou zobrazeny na Obrázku 4.



Obrázek 4 - CFG k funkci `calculation`

4 Analýza vstupního programu

Úkolem implementovaného programu je zachytit vztahy mezi hodnotami proměnných v každém bodu programu. V článku [2] autoři navrhli algoritmy pro intraprocedurální a interprocedurální analýzy, které jsou určeny pro interpretaci zadání s lineárními výrazy na pravé straně. S ohledem na skutečnost, že lineární výrazy lze vhodným způsobem zachytit v podobě vektorů a matic, pracují navržené algoritmy právě s těmito strukturami.

Hodnoty proměnných v jednom bodu programu v jednom okamžiku lze zachytit vektorem. Pro zachycení hodnot proměnných v daném bodu programu ve všech běžích programu vznikne množina vektorů. Tato množina bude velmi pravděpodobně obsahovat velké množství vektorů, proto je třeba nalézt způsob, jak tuto množinu reprezentovat. K tomu se používají tzv. množiny generátorů, tedy vektorů, které reprezentují všechny vektory, které obsahují hodnoty, kterých mohou proměnné v daném programovém bodu nabývat.

Pro vyjádření změn, které provádí každý jeden příkaz vstupního programu, využívá analýza matice. Pokud je příkazem přiřazení s lineárním výrazem na pravé straně, pak je do jednotkové matice na řádek, odpovídající proměnné na levé straně přiřazení, přepsán lineární výraz. Pro tento přepis jsou přesně definována pravidla, která jsou popsána dále v Kapitole 4.1. Pokud na pravé straně přiřazení není lineární výraz, pak jsou pro dané přiřazení sestaveny 2 matice, kdy jedna odpovídá tomu, že je proměnná na levé straně přiřazení rovna nule, a druhá odpovídá tomu, že je tatáž proměnná rovna jedné. Podrobně je sestavení a použití vektorů a matic popsáno v následující kapitole.

Interprocedurální analýza, která je hlavním úkolem programu, zpracovává vstupní program převedený do samostatných CFG pro každou funkci programu. CFG jsou procházeny ve třech samostatných průchodech. Při prvním průchodu jsou analyzovány všechny CFG zároveň a výstupem tohoto průchodu je množina matic, zachycující změny hodnot proměnných, které každá funkce realizuje. Při druhém průchodu algoritmus využívá množiny matic vypočtené v prvním průchodu k tomu, aby správně simuloval volání jednotlivých funkcí. Výstupem druhého průchodu jsou vypočtené množiny vektorů generující všechny možné hodnoty, kterých mohou nabývat lineární vztahy v jednotlivých bodech programu. Na závěr algoritmus projde všechny body programu a vyhodnocením vypočtených množin vektorů získá konečné hodnoty lineárních vztahů, vyjadřujících vztahy mezi proměnnými, v každém bodu vstupního programu.

Pro zaznamenání vztahů v druhém průchodu algoritmu se vychází z možnosti zachytit lineární vztahy pomocí matic a vektorů, proto jsou hodnoty jednotlivých proměnných v těchto vztazích ukládány do vektorů, resp. pomocí množin generátorů. Pravidla pro práci s generátory jsou popsána v následující kapitole. Stejná pravidla, která platí pro práci s vektory a generátory, lze uplatnit i pro práci s maticemi pro sestavení množiny matic reprezentující změny prováděné konkrétní funkcí.

Při výpočtech generátorů postupuje program tak, že vždy přečte příkaz, pomocí kterého má vstupní program přejít do následujícího stavu a sestaví z něj matici zachycující změnu, kterou daný příkaz na sledovaných proměnných provádí. Sestavenou maticí pak zleva vynásobí každý generátor vypočtený v posledním zpracovaném stavu vstupního programu. Z nových vypočtených vektorů pak sestavuje novou množinu generátorů pro aktuálně zpracovávaný stav vstupního programu. Pravidla definovaná pro vytváření množin generátorů zajišťují, že do množiny jsou vždy přidávány pouze takové vektory, které ji rozšíří o nové hodnoty.

Pokud je dalším příkazem pro přechod do následujícího stavu vstupního programu volání funkce, pak se generátory vypočtené v posledním zpracovaném stavu vstupního programu zleva vynásobí postupně všemi maticemi z množiny matic, která je výstupem volané funkce.

Výstupem programu je pak výpis vstupního programu, doplněný o výpis vypočtených vztahů mezi hodnotami, kterých mohou jednotlivé globální proměnné nabývat v každém bodu programu.

4.1 Sestavení vektorů a matic

Na počátku procházení vstupního programu jsou načteny globální proměnné. Podle počtu proměnných je sestavena množina generátorů pro vstupní bod analyzovaného programu. Tato množina se vždy skládá z jedničkových vektorů, jejichž počet je dán právě počtem proměnných analyzovaného vstupního programu. Pokud má analyzovaný program N globálních proměnných, pak na vstupním bodu analyzovaného programu bude inicializována množina $N + 1$ jednotkových vektorů. Například pro vstupní bod analyzovaného programu, který bude mít dvě globální proměnné, bude vytvořena množina generátorů

$$\left\{\begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\end{bmatrix}\right\}.$$

Následně začne procházení jednotlivých příkazů analyzovaného programu. Pro jednotlivé načtené příkazy je třeba sestavit matice zachycující změny prováděné těmito příkazy. Pro vytváření těchto změnových matic je třeba zavést následující pravidla.

Nechť $X = \{x_1, \dots, x_k\}$ je množina globálních proměnných vstupního programu. Jelikož navržené algoritmy jsou určeny pouze pro zpracování afinních přiřazení, tj. přiřazení ve tvaru

$$x_j = t_0 + \sum_{i=1}^{k} t_i x_i, \text{ pro } t_i \in \mathbb{Z}_m,\ i = 0, \dots, k, x_j \in X, \tag{4.1}$$

nebo nedeterministická zadání $x_j = ?$ (což jsou všechny ostatní příkazy, které nesplňují předchozí definici), lze způsob, jakým přečtený příkaz ovlivní globální proměnné, zachytit pomocí matice. Jak je známo z lineární algebry, pravou stranu uvedeného afinního přiřazení (4.1) lze zapsat v podobě vektoru $[t_0, t_1, \dots, t_k]$, kde platí, že každému t_i je přiřazeno právě jedno x_i, pro $i > 0$. Speciální případ je t_0, kterému dle výše uvedeného vztahu žádné x_0 přiřazeno není, ale je třeba jeho hodnotu také zohlednit. Z tohoto důvodu je vždy uvažováno $x_0 = 1$. Lineární transformaci, způsobenou uvedeným afinním zadáním (4.1), lze popsat maticí

$$\left[\begin{array}{c|c} I_j & 0 \\ t_0 \dots t_{j-1} & t_j \dots t_k \\ 0 & I_{k-j} \end{array}\right],$$

kde I_j je jednotková matice. Změna se na aktuální hodnoty vektorů, které byly vypočteny pro bod, který zachycené změně předcházel, projeví jako násobení vypočteného vektoru zleva maticí zachycující změnu proměnné x_j.

Nechť pro příklad je v aktuálně zpracovávaném bodu vstupního programu uložena množina vektorů obsahující jediný vektor $[x_0, x, y]^t = [7, 8, 9]^t$ a je načteno přiřazení $x_1 = 5 + 4x - 3y$. Pokud $[x_0, x_1, y_1]^t$ bude vektor po zpracování přečtené změny, pak

$$\begin{bmatrix} x_0 \\ x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 5 & 4 & -3 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix}.$$

Jelikož násobení matic je asociativní, je možné postupným násobením zleva sloučit více matic do jedné a tímto způsobem zachytit více změn najednou. Pokud M_i, pro $i = \{1, \dots, k\}$, je matice zachycující postupně první až k-tou změnu realizovanou vstupním programem a M je matice zachycující všech k změn dohromady, pak

$$M = M_k * M_{k-1} * \dots * M_1.$$

V případě, že jsou globální proměnné inicializované, provede program předtím, než začne zpracovávat jednotlivé funkce vstupního programu, zpracování inicializace proměnných. To probíhá stejným způsobem, jako zpracování všech ostatních příkazů ve funkcích. Pokud má analyzovaný program dvě globální proměnné a je jako první načtena inicializace proměnné $x = 2$, pak je sestavena matice změny

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

a touto maticí jsou zleva vynásobeny postupně všechny vektory obsažené ve vstupním bodu (tj. jednotkové vektory). Tím je získána nová množina vektorů

$$\left\{ \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\},$$

ze které bude sestavena, dle níže popsaného algoritmu, nová množina generátorů pro aktuální bod analyzovaného programu. Pro vstupní bod funkce `main` je potom jako výchozí množina použita množina generátorů, která vznikne po přečtení inicializace proměnných.

4.2 Výpočty v modulární aritmetice

Základní definice modulární aritmetiky říká, že čísla $a, b \in \mathbb{Z}$ jsou kongruentní modulo m, pro nějaké $m \in \mathbb{N}$, jestliže $(a - b)$ je dělitelné m. To znamená, že $a \bmod m = b \bmod m$. Jinak řečeno, v modulární aritmetice jsou čísla se stejným zbytkem po dělení shodná. Díky tomu vznikají tzv. *zbytkové třídy*, obsahující vždy čísla se shodným zbytkem po dělení. Zbytková třída čísla a vzhledem k relaci kongruence modulo m se značí $[a]_m = \{b \mid b \equiv a \ (mod\ m)\}$ [5]. Množina celých čísel $\mathbb{Z}$ se prostřednictvím zbytkových tříd rozpadá na přesně m zbytkových tříd a vzniká *množina zbytkových tříd* $\mathbb{Z}_m = \{[a]_m \mid a \in \mathbb{Z}\}$ [5].

Na množině $\mathbb{Z}_m$ jsou definovány operace sčítání ($[a]_m + [b]_m = [a + b]_m$) a násobení ($[a]_m * [b]_m = [a * b]_m$). Operace sčítání a násobení jsou komutativní a asociativní. Operace násobení je vůči operaci sčítání distributivní. Operace sčítání má definovaný inverzní prvek $-[a]_m = [m - a]_m$. Operace násobení má inverzní prvek k $[a]_m$ právě tehdy, když a a m jsou nesoudělné.

Z výše uvedeného vyplývá, že pro každé m lze vždy pracovat pouze s množinou čísel $\{0, 1, \dots, m - 1\}$.

Tato práce se zabývá analýzou v modulární aritmetice pro mocniny 2. Důvodem je, že běžné programovací jazyky provádí aritmetické operace pro celočíselné typy modulo $m = 2^w$, kde např. $w = 32$ pro datový typ int, $w = 64$ pro datový typ long. Dále tedy bude práce pojednávat pouze o okruhu zbytkových tříd $\mathbb{Z}_m$, kde $m = 2^w$, $w \geq 1$.

V tomto případě platí některá pravidla, která v obecné modulární aritmetice uplatnit nelze. Předpokládejme, že $a \in \mathbb{Z}_m, a \neq 0$. Pak platí [2]:

a) Pokud je a sudé, pak je a dělitel nuly, tj. platí $a * b \equiv 0 \, (mod\, m)$ pro nějaké $b \in \mathbb{Z}_m, b \neq 0$.
b) Pokud je a liché, pak je a invertibilní, tj. platí $a * b \equiv 1 \, (mod\, m)$ pro nějaké $b \in \mathbb{Z}_m$.
c) *Hodnost*[2] čísla a, značí se r, je definována jako $r \in \{0, \dots, w\}$, jestliže $a = 2^r * a'$ pro nějaký invertibilní prvek a'. Platí, že $r = 0$, pokud je a invertibilní, a $r = w$, pokud $a = 0 \, (mod\, m)$.

Jak bylo vysvětleno v předchozí kapitole, navržený program pracuje s vektory a maticemi. Proto je třeba definovat ještě několik dalších vlastností [2].

Množina M vektorů $[x_1, \dots, x_N]^t$ s prvky x_i ze $\mathbb{Z}_m$ je $\mathbb{Z}_m$*-modulem* právě tehdy, když $M \subseteq \mathbb{Z}_m^N$, a M je uzavřená vůči vektorovému součtu a skalárnímu součinu. Množina G je *množinou generátorů* M právě tehdy, když $G \subseteq M$ a když $M = \left\{\sum_{i=1}^{l} r_i g_i \mid l \geq 0, r_i \in \mathbb{Z}_m, g_i \in G\right\}$, tedy že všechny prvky M jsou lineárními kombinacemi prvků G. Pak je M je *generováno* G, což lze zapsat $M = \langle G \rangle$.

Každý nenulový vektor $x = [x_1, \dots, x_N]^t$ má *vedoucí index* i, pro který platí $x_i \neq 0$ a $x_{i'} = 0$ pro všechna $i' < i$. Prvek x_i vektoru x se nazývá *vedoucí prvek* x. Množina G je ve *schodovitém tvaru*, jestliže pro všechny vektory $x \in G$ platí, že x je nenulový vektor a vedoucí indexy všech $x \in G$ jsou rozdílné.

Při rozšiřování množiny G o vektor x jsou nejprve porovnány vedoucí indexy všech vektorů $x' \in G$ s vedoucím indexem vektoru x. Pokud jsou odlišné, znamená to, že G neobsahuje vektor se stejným vedoucím indexem a prostým přidáním vektoru x do množiny G nedojde k porušení schodovitého tvaru G. Pokud však v množině G již existuje vektor se stejným vedoucím indexem, jako je vedoucí index vektoru x, je třeba zavést následující pravidla.

Nechť i je vedoucí index pro nějaký vektor $y \in G$. Každé $x \in \mathbb{Z}_m$ lze zapsat jako

$$x = d * 2^r, \text{ kde } d \text{ je invertibilní.} \tag{4.2}$$

Potom vedoucí prvek y_i lze zapsat jako $y_i = d' * 2^{r'}$ a obdobně vedoucí prvek x_i lze zapsat jako $x_i = d * 2^r$.

Pokud $r' \leq r$, pak je možné vynulovat vedoucí prvek x_i. Tím vznikne nový vektor x', který lze získat pomocí vzorce

$$x' = d' * x - 2^{r-r'} * d * y. \tag{4.3}$$

Výpočet vektoru x' se nazývá *redukční krok* a výpočet pokračuje dále s množinou generátorů G a nově vypočteným vektorem x'.

V případě, kdy $r' > r$ je třeba nejprve vyměnit vektory x a y tak, že vektor x se stane součástí množiny generátorů G, čímž vznikne upravená množina generátorů G', a vektor y bude

[2] Anglicky dle [2] „rank“

vkládán. Pokud zůstane zachováno značení uvedené výše, tedy že y je označován vektor z G a x je označen vektor vkládaný do G, pak po výměně vektorů lze provést vynulování vedoucího prvku nově vkládaného vektoru opět podle vzorce (4.3). Další výpočet opět pokračuje s nově vypočteným vektorem x' a množinou generátorů G'. (Strukturovaný popis algoritmu je uveden v příloze II. Algoritmus dále označován jako ALG-II.1.)

Algoritmus ALG-II.1 tak, jak je popsán, může mít však jednu vadu, kterou lze nejlépe popsat na příkladu [2]. Mějme množinu $G = \{[8, 1, 3]^t\}$, do které by měl být přidán vektor $x = [0, 2, 6]^t$. Na první pohled lze vektor x do generátoru přidat, neboť vedoucí index vektoru x není v množině G v žádném vektoru. Na druhý pohled je však zřejmé, že vektor x je násobkem vektoru obsaženého v množině G a tudíž by do množiny G neměl být přidán.

Aby se tomuto problému předešlo, je třeba zajistit, aby množina G nereprezentovala pouze konkrétní vypočtené vektory, ale i všechny jejich lineární kombinace. Množina obsahující všechny lineární kombinace všech svých prvků se nazývá *saturovaná množina*. Tato definice v případě množiny G, která má omezený počet prvků kvůli schodovitému tvaru, ve kterém je udržována, znamená, že musí obsahovat vektory, které reprezentují všechny lineární kombinace prvků množiny G. Množina G ve schodovitém tvaru je tedy saturovaná právě tehdy, když vektor $x' = 2^{(w-r)} * x$ je *redukovatelný* s ohledem na $G\backslash\{x\}$ pro každé $x \in G$, kde r je hodnost vedoucího prvku x. Vektor x' je redukovatelný právě tehdy, když množina $\bar{G}$ vypočtená pomocí algoritmu ALG-II.1 pro sjednocení G a vektoru x, je shodná s G, což znamená, že aplikací redukčních kroků, popsaných výše, výhradně na vektor x, vznikne nulový vektor. Pokud množina G nebude saturovaná, algoritmus ALG-II.1 výše popsaný problém neodhalí.

Pokud již množina G saturovaná je, pak pro ověření, že saturovaná zůstane i po přidání nového vektoru x, je třeba před vložením vektoru x vypočíst vektor

$$x' = 2^{w-r} * x, \text{ kde } r \text{ je vypočteno z vedoucího prvku vektoru } x. \tag{4.4}$$

Pokud x' je nenulový vektor, pak se opakuje postup pro vložení vektoru x' do G. Pokud x' je nulový vektor, pak po vložení vektoru x zůstane množina G saturovaná. Z výše uvedených definic pro modulární aritmetiku modulo 2^w je zřejmé, že pokud bude vedoucí prvek vektoru x invertibilní (tedy lichý), pak bude $r = 0$ a tudíž bude vektor x násoben 2^w, což vždy odpovídá 0. Proto je výpočet (4.4) nutné provádět pouze v případě, že vedoucí prvek vkládaného vektoru je sudý.

Postup popsaný pro sestavování a udržování množiny generátorů, lze obdobně použít i pro matice zachycující změny v jednotlivých krocích vstupního programu. Pro použití postupu popsaného pro vektory lze matici o rozměru $n \times n$ reprezentovat jako vektor o n^2 prvcích.

4.2.1 Vyhodnocení homogenní soustavy lineárních rovnic

Při závěrečném průchodu všemi CFG bude třeba vyhodnotit vztahy, zachycené množinami generátorů v jednotlivých uzlech CFG tak, aby byly získány konečné hodnoty proměnných v odpovídajících stavech vstupního programu. Za tímto účelem je třeba vypočíst homogenní soustavu lineárních rovnic, která je zachycena v každé množině generátorů v každém uzlu CFG. Nechť řešená soustava lineárních rovnic je označena

$$Ax = 0. \tag{4.5}$$

Z předchozího textu vyplývá, že pro N proměnných, může mít tato soustava maximálně N rovnic (je uvažována i přidaná proměnná x_0 pro přičítané konstanty v lineárních vztazích). Přidáním případných dalších rovnic s nulovými koeficienty lze předpokládat, že soustava bude mít přesně N rovnic. Pak ve vztahu (4.5) je A čtvercová matice o $N \times N$ prvcích, s prvky $a_{ij} \in \mathbb{Z}_m$, kde $1 \leq i,j \leq N$, $x = [x_1, \dots, x_N]^t$ a 0 je nulový sloupcový vektor.

Pro nalezení všech řešení vztahu (4.5), množina všech řešení se značí $\mathbb{L}_A$, je třeba nejprve převést matici A do diagonálního tvaru. Postup pro převedení matice s prvky v $\mathbb{Z}_m$ do diagonálního tvaru je podobný, jako u matice s prvky např. v $\mathbb{R}$ (obecně v nějakém tělese). Hlavní rozdíl je způsoben tím, že zatímco v $\mathbb{R}$ platí, že každý nenulový prvek je invertibilní, v $\mathbb{Z}_m$, kde $m = 2^w$, jsou invertibilní pouze lichá čísla.

Při úpravě matice A, s prvky ze $\mathbb{Z}_m$, je třeba nejprve vyhledat *pivotní člen*. Pro každý člen matice lze vypočíst rozklad dle (4.2). Jako pivotní člen je vždy vybrán prvek v řádku i a sloupci j s nejmenší hodností r. Dle (4.2) platí, že pivotní člen $p = d * 2^r$. Následně mohou být všechny prvky v řádku i a sloupci j, s výjimkou pivotního členu, vynulovány, neboť minimální r zajišťuje, že všechny ostatní prvky v řádku i a sloupci j jsou násobky 2^r. Vynulování všech prvků v řádku i a sloupci j se provádí vynásobením odpovídajícího řádku nebo sloupce pomocí d (tedy $x' = x * d$, kde x' je nová hodnota prvku x) a následně odečtením vhodného násobku řádku i nebo sloupce j. Pokud prvek, který má být vynulován bude označen $x = d' * 2^s$ a hledaný vhodný násobek α, pak:

$$x' = d * x = d * d' * 2^s = d * d' * 2^{s-r} * 2^r = (d * 2^r) * (d' * 2^{s-r}) = p * \alpha$$

Z uvedeného rozkladu tedy vyplývá, že vhodný násobek

$$\alpha = d' * 2^{s-r}. \tag{4.6}$$

Jelikož hodnota r (resp. s) odpovídá počtu nul na konci binárního zápisu čísla, tak rozdíl $s - r$ v exponentu vyjadřuje bitový posun čísla x o r bitů vpravo.

Po vynulování všech prvků ve sloupci a řádku s pivotním členem dojde výměnou sloupců a řádků k přemístění pivotního členu do diagonály matice a pokračuje se se zbylou částí matice. Pro úpravu matice do diagonálního tvaru tedy mohou být provedeny následující úpravy:

a) násobení řádku nebo sloupce lichým číslem d, kde d pochází ze vztahu $x = d * 2^r$, kde x je zvolený pivotní člen
b) přičítání (resp. odčítání) násobku jiného řádku nebo sloupce
c) výměna dvou sloupců nebo řádků

Graficky je úprava matice na diagonální naznačena na Obrázku 5. Tmavší odstín barvy vždy znázorňuje vybraný pivotní člen a světlejší značí prvky, které musí být pomocí úprav vynulovány. Nevybarvený prostor pak vždy znázorňuje, v jaké části matice je pivotní člen vyhledáván.

Obdobně, jako u úprav matic v $\mathbb{R}$ se zaznamenávají řádkové a sloupcové transformace formou shodných úprav na jednotkových maticích, lze i v $\mathbb{Z}_m$ rozložit matici A na

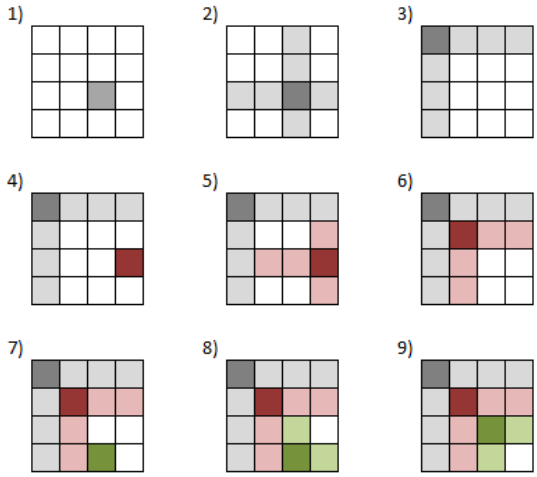
$A = L * D * R$, kde D je diagonální matice a L, R jsou $(N \times N)$-matice invertibilní v $\mathbb{Z}_m$. (4.7)

Matice L zaznamenává převrácené hodnoty řádkových transformací prováděné při úpravách A na D a matice R zaznamenává převrácené hodnoty transformací sloupcových. Na vztahy (4.5) a (4.7) lze aplikovat standardní matematické úpravy, tedy $Ax = 0$ lze zapsat jako $LDRx = 0$, dále

lze celou rovnici vynásobit L^{-1} a v získaném tvaru $DRx = 0$ je možné položit $Rx = x'$. Pak platí, že

$$Dx = 0 \text{ právě tehdy, když } x = R^{-1}x' . \tag{4.8}$$

S ohledem na skutečnost, že matice R zaznamenává převrácené hodnoty sloupcových transformací, pak matici R^{-1} lze získat aplikací sekvence sloupcových operací, které byly provedeny na matici A, na jednotkovou matici.



Obrázek 5 - Grafické znázorněné úpravy matice na diagonální

Pokud je matice A upravena do diagonálního tvaru, pak pro její diagonální prvky dle (4.2) platí $a_{ii} = d_i * 2^{r_i}$, kde d_i je invertibilní. Potom platí, že množina řešení soustavy rovnic (4.5) je $\mathbb{Z}_m$-modul generovaný z vektorů

$$l_j = 2^{w-r_i} * e_j, \text{ kde } j = \{1, \dots, N\} \text{ a } e_j \text{ je } j\text{-tý jednotkový vektor.} \tag{4.9}$$

Algoritmus vyhodnocující získanou soustavu rovnic tedy nejprve provádí úpravu matice A na diagonální a přitom všechny sloupcové operace zároveň provádí na jednotkové matici. Po dokončení výpočtu diagonální matice D s diagonálními prvky a_{ii} lze vypočíst vektory $l_1, \dots, l_N$ dle vztahu (4.9). Množina těchto vektorů generuje řešení soustavy $Dx = 0$. Pro získání množiny vektorů generující řešení soustavy $Ax = 0$ je třeba vypočíst $x = [x_1, \dots, x_N]^t$ aplikací vzorce (4.8), tj. $x_j = R^{-1}l_j$, pro $j = \{1, \dots, N\}$.

4.3 Interprocedurální analýza

Princip interprocedurální analýzy byl již popsán v Kapitole 2. Cílem této kapitoly je popsat interprocedurální analýzu v podmínkách implementovaného algoritmu. V rámci navržené interprocedurální analýzy dochází k procházení CFG všech funkcí postupně ve třech samostatných cyklech. V prvním dojde k vypočtení množin změnových matic na výstupech všech funkcí a k jejich distribuci do programových bodů funkcí, ze kterých byly volány.

V druhém cyklu jsou vypočítány množiny generátorů generující hodnoty pro lineární vztahy platné v daném bodu programu, ze kterých jsou následně ve třetím cyklu vypočteny konečné hodnoty afinních relací, vyjadřujících vztahy mezi proměnnými v jednotlivých bodech programu. Pro každý z uvedených cyklů je definován algoritmus, který ho zpracovává [2].

Procházení CFG je realizováno metodou prohledávání do šířky. Pokud při procházení dojde algoritmus do vrcholu, kde se CFG větví kvůli nějaké podmínce, prochází všechny větve bez ohledu na vyhodnocení podmínky. Autoři [2] dokazují, že zohlednění podmínek větvení při interprocedurální analýze afinních programů je časově náročné (konkrétně dokazují, že časová složitost je DEXPTIME-úplná), a navrhují i metody pro přibližnou práci s podmínkami větvení. Těmi se však tato práce nezabývá.

Algoritmus prvního cyklu nejprve vytvoří prázdnou frontu pro ukládání vrcholů, které mají být zpracovány. Vrcholy se do fronty ukládají jako uspořádané dvojice (u, M), kde u je vrchol který byl změněn a M je matice zachycující změnu, která vyvolala změnu vrcholu u. Následně je do všech vrcholů ve všech CFG uložena prázdná množina (později se do této množiny budou ukládat matice zachycující změny pro daný vrchol) s výjimkou vstupních vrcholů všech CFG, do kterých jsou uloženy čtvercové jednotkové matice rozměru $(N + 1) \times (N + 1)$, kde N je počet globálních proměnných analyzovaného programu. Všechny vstupní vrcholy jsou zároveň vloženy do fronty vrcholů ke zpracování. Následně je z fronty odebrána a zpracována první uspořádaná dvojice (u, M). Pokud má vrchol u nějaké výstupní hrany s, které vedou do vrcholu v, pak algoritmus vypočte pro každou hranu s množinu new, která obsahuje výsledky násobení $M_1 * M$, kde M_1 je matice z množiny generující změnové matice pro přechod hranou s do uzlu v. Dále je pomocí algoritmu ALG-II.1 postupně každá matice z množiny new vložena do množiny matic zachycujících změny pro vrchol v, označené $S(v)$. Pro umožnění použití uvedeného algoritmu je po dobu zpracování tímto algoritmem matice $n \times n$ převedena na sloupcový vektor o n^2 prvcích. Pokud dojde ke změně množiny $S(v)$, musí být na konec fronty vrcholů ke zpracování přidána nová uspořádaná dvojice (v, M_2), kde $M_2 \in new$ je matice, která způsobila změnu množiny $S(v)$.

Pokud je vrchol u výstupním uzlem nějaké funkce q (volání funkce q slouží k přechodu mezi uzly u' a v, které oba náležejí do nějaké funkce r), pak je maticí M zleva vynásobena každá matice $M_1 \in S(u')$ a výsledek je uložen do množiny new. Následně jsou všechny matice $M_2 \in new$ vkládány do množiny $S(v)$ pomocí algoritmu ALG-II.1. Pokud dojde k přidání nějaké matice M_i do množiny $S(v)$, je na konec fronty vrcholů ke zpracování přidána nová uspořádaná dvojice (v, M_2). (Strukturovaný popis algoritmu je uveden v příloze II.)

Následně je třeba uložit na hrany, obsahující volání nějaké funkce, množinu matic z výstupního uzlu volané funkce, která zachycuje všechny změny prováděné volanou funkcí.

Algoritmus druhého cyklu pracuje na obdobném principu jako algoritmus prvního cyklu. Nejprve všem vrcholům u všech CFG nastaví množinu generátorů $R(u)$ jako prázdnou a vstupní vrchol CFG funkce main inicializuje jednotkovými vektory, jejichž počet je $N + 1$ pro N globálních proměnných analyzovaného programu. Dále vytvoří frontu vrcholů ke zpracování, kterou naplní uspořádanými dvojicemi (u, x), kde u je vrchol a x je vektor, pro všechny vektory vstupního vrcholu funkce main. Následně algoritmus postupně odebírá dvojice (u, x) z fronty a pro každou odebranou dvojici zkoumá vždy všechny hrany s, které vedou z u do nějakého v. Pokud je na hraně s uloženo volání nějaké funkce q, pak musí být vektor x přidán do množiny generátorů vstupního vrcholu funkce q (označen st_q) pomocí ALG-II.1 a pokud dojde ke změně množiny $R(st_q)$, pak je do fronty přidána dvojice (st_q, x). Dále dochází k výpočtu množiny vektorů new, která obsahuje výsledky výpočtů $M * x$ a každý vektor $x' \in new$ je pak vkládán do množiny generátorů $R(v)$ pomocí algoritmu ALG-II.1. Pro každý vektor x', který způsobí změnu množiny $R(v)$ je vytvořena uspořádaná dvojice (v, x'), která je vložena na konec fronty.

Stejně jako v prvním cyklu, i v tomto případě se algoritmus opakuje, dokud není fronta prázdná. (Strukturovaný popis algoritmu je uveden v příloze II.)

Algoritmus třetího cyklu prochází postupně všechny vrcholy všech CFG a v každém z nich provádí vyhodnocení lineárních vztahů zachycených v množinách generátorů. Pro vyhodnocení je tedy každá množina generátorů převedena na čtvercovou matici tak, že případné chybějící vektory jsou nahrazeny nulovým vektorem. Při převodu jsou jednotlivé vektory z množiny generátorů vkládány do matice jako řádky této matice. Po sestavení matice může dojít k vyhodnocení homogenní soustavy lineárních rovnic, jak byla popsána ve stejnojmenné kapitole.

Algoritmus tedy vypočte rozklad pro všechny prvky matice dle vzorce (4.2) a nalezne prvek $p \neq 0$ s nejnižším r (tzv. pivot) a uloží jeho souřadnice. Pokud takových prvků existuje více, zvolí ten, který je nalezen jako první. Po té pro každý prvek x ve sloupci s pivotem vypočte hodnotu α dle (4.6) a následně pro každý prvek x' v řádku s x vypočte rozdíl $x' * d$ (kde d je vypočteno dle vztahu (4.2) z pivotu) a $p' * \alpha$ (kde p' je prvek z řádku s pivotem se shodným sloupcovým indexem, jako má prvek x'). Tímto postupem dojde k vynulování všech prvků ve sloupci s pivotem. Obdobný postup se opakuje pro všechny prvky v řádku, kdy úpravy probíhají na sloupcích. Pro výpočet vzorců (4.7), resp. (4.8) je nutné všechny úpravy, prováděné na sloupcích matice, zachytávat na jednotkové matici, proto všechny úpravy prováděné při nulování prvků v řádku pivotu, jsou zároveň realizovány na jednotkové matici stejného rozměru, jaký má řešená matice. Po vynulování všech prvků v řádku i sloupci s pivotem je řádek s pivotem umístěn na první řádek matice (původní první řádek matice je umístěn na dosavadní pozici řádku s pivotem) a sloupec s pivotem je umístěn do prvního sloupce matice (analogicky je i původní první sloupec matice přesunut na dosavadní pozici sloupce s pivotem). Výměna sloupců se stejným způsobem provede i na upravené jednotkové matici. Tím se pivot umístí na souřadnice $(1,1)$ řešené matice a celý postup se opakuje pro zbylou část matice. První řádek a první sloupec řešené matice jsou již zpracovány a do dalších výpočtů nejsou zahrnuty.

Po dokončení výpočtů je výstupem diagonální matice D a matice R^{-1} zachycující sloupcové transformace (prováděné při výpočtu D) na jednotkové matici. Aplikací vztahů (4.8) a (4.9) na jednotlivé prvky diagonály matice D jsou nakonec vypočteny vektory generující řešení soustavy lineárních rovnic. Jejich uspořádání do saturované množiny ve schodovitém tvaru je opět zajištěno pomocí algoritmu ALG-II.1. (Strukturovaný popis algoritmů pro výpočet diagonální matice i soustavy $Ax = 0$ je uveden v příloze II.)

Z popisu algoritmů je zřejmé, že uzly ke zpracování se do fronty přidávají pouze v případě, že dojde ke změně jejich obsahu, čímž je zajištěno, že algoritmus vždy skončí.

5 Implementace

Tato kapitola se nejprve zabývá návrhem implementace navržených algoritmů, které byly představeny v předchozí kapitole a následně i popisem samotné implementace aplikace. V rámci návrhu dojde k rozdělení aplikace do dvou v podstatě samostatných celků, které budou v rámci implementace dále členěny.

Implementace byla provedena v programovacím jazyce C# a pro vývoj byl použit nástroj Microsoft Visual Studio 2010. Jelikož úkolem aplikace je téměř výhradně zpracování vstupního textového souboru s programem a kromě zadání vstupních parametrů neumožňuje žádnou součinnost uživatele, jeví se jako zbytečné implementovat aplikaci s jakýmkoli grafickým rozhraním. Z tohoto důvodu byla aplikace realizována jako konzolová aplikace s výstupem do textového souboru a volitelnou možností zobrazení průběžných stavů aplikace na konzoli.

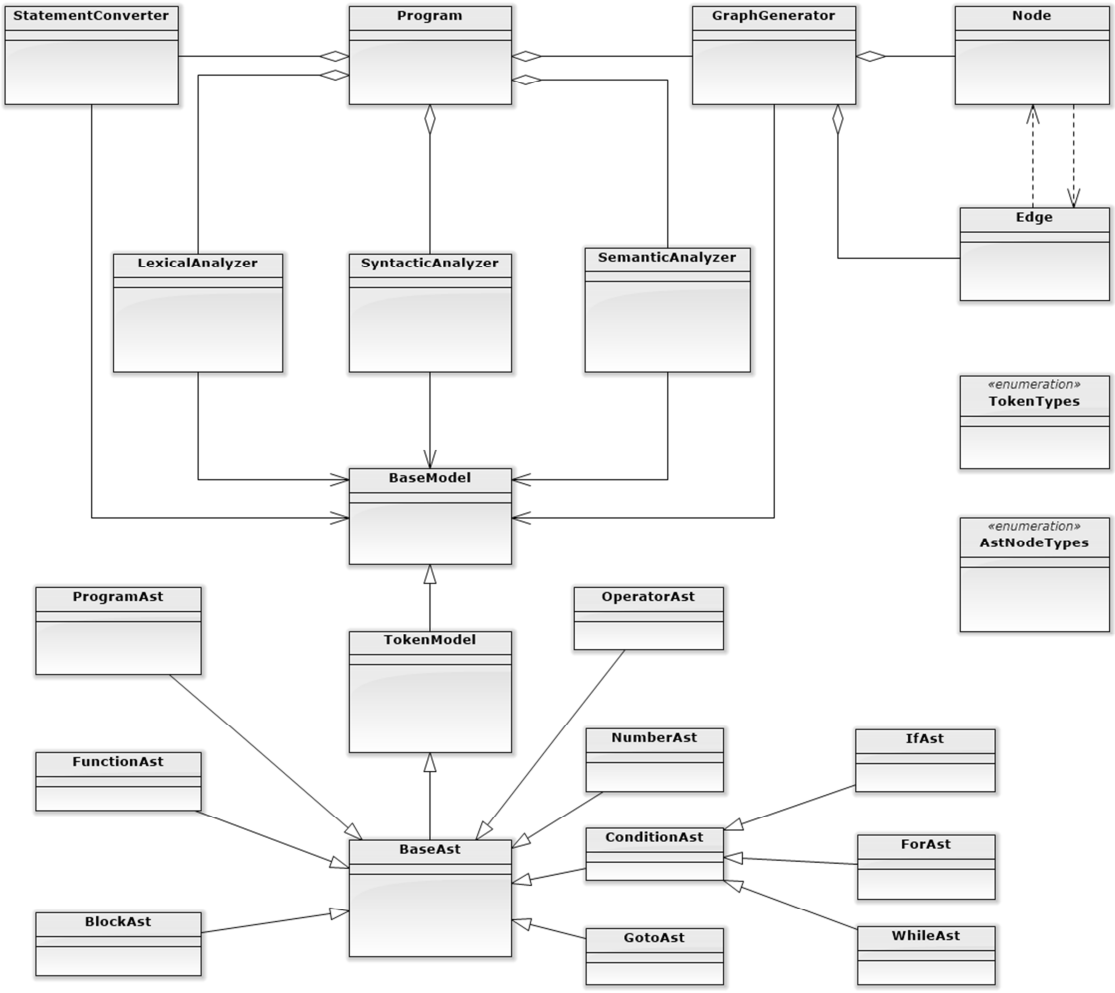
5.1 Návrh implementace

Aplikace pro provedení interprocedurální analýzy dle [2] je navržena jako konzolová aplikace, jejímž vstupem je analyzovaný program. Úkolem aplikace je nejprve načíst analyzovaný program a zkontrolovat, zda je správná jeho syntaxe. Následně musí převést jednotlivé funkce analyzovaného programu do struktury CFG, a po té teprve může provést požadovanou analýzu. Návrh aplikace je rozdělen do dvou částí, kdy jedna část se zabývá zpracováním vstupního textového souboru s analyzovaným programem a druhá část pak provádí samotnou analýzu. Schéma první části aplikace je zachyceno na zjednodušeném třídním diagramu (Obrázek 6), schéma druhé části pak na Obrázku 7. Kompletní třídní diagram je v příloze č. III.

5.1.1 Zpracování vstupního textového souboru

V první fázi své činnosti musí aplikace načíst textový soubor obsahující zdrojový kód analyzovaného programu a zkontrolovat, zda syntaxe programu odpovídá pravidlům definované bezkontextové gramatiky. K tomu slouží třídy `LexicalAnalyzer`, `SyntacticAnalyzer` a `SemanticAnalyzer`.

Jak již názvy napovídají, úkolem třídy `LexicalAnalyzer` je provedení lexikální analýzy, zatímco úkolem třídy `SyntacticAnalyzer` je provedení syntaktické analýzy. Třída `SemanticAnalyzer` je určena k provedení zjednodušené sémantické analýzy v rozsahu důležitém pro potřeby implementované aplikace. Jejím úkolem je tedy pouze provedení kontroly, zda byly všechny volané funkce řádně deklarovány. Dále je implementována třída `StatementConverter`, která slouží ke konverzi načteného programu na úroveň posloupnosti příkazů řízených pouze příkazy „if“ a „goto-label“. Třída `GraphGenerator` má za úkol převod vytvořené posloupnosti příkazů na CFG. Třída `BaseModel` a všechny třídy z ní dědící slouží k rozlišení a určení typů tokenů vytvářených v lexikální analýze. Návrh počítá také s definováním dvou výčtových typů sloužících pro určení typu uzlu v AST (výčtový typ `AstNodeTypes`) a pro určení typů tokenů (výčtový typ `TokenTypes`).

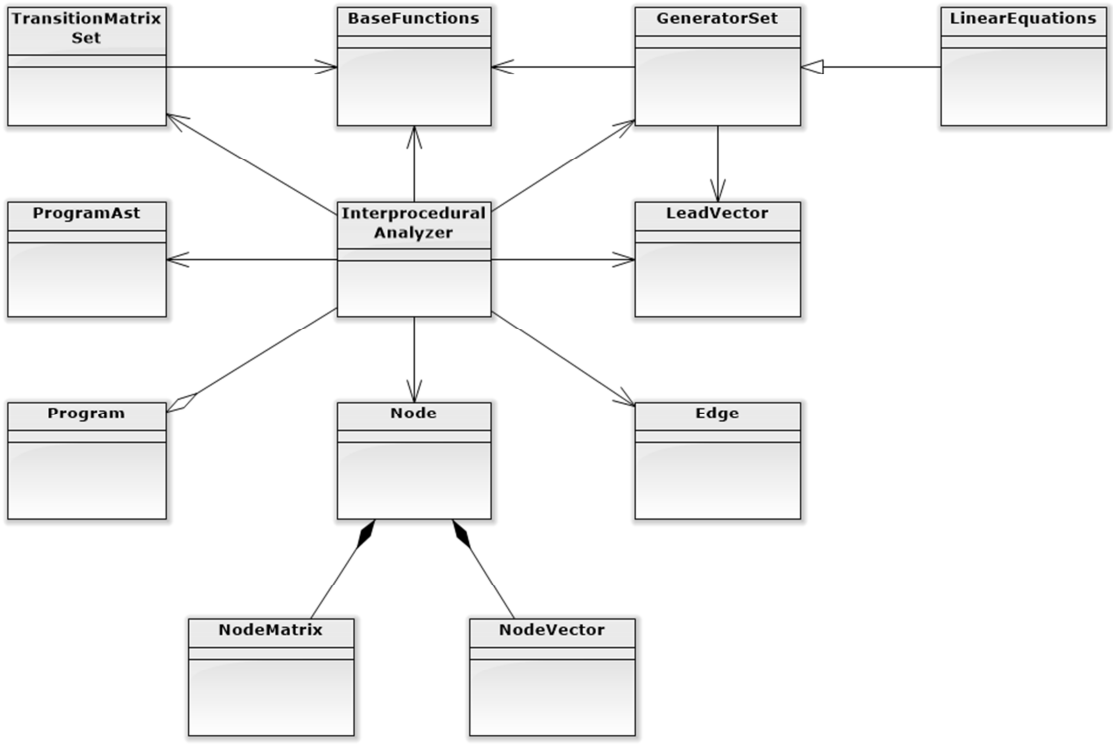


Obrázek 6 - Zjednodušený třídní diagram části aplikace pro zpracování vstupního textového souboru

5.1.2 Interprocedurální analýza načteného programu

Po té, co je načtený text zpracován do CFG, může být zahájena samotná analýza. Interprocedurální analýza především implementuje algoritmy popsané v kapitole Interprocedurální analýza a v příloze č. II. Samotné algoritmy navržené interprocedurální analýzy jsou ve třídě `InteproceduralAnalyzer` s využitím pomocných tříd jako např. `GeneratorSet` pro uložení množin generátorů v uzlech CFG, `TransitionMatrixSet` pro uložení množin změnových matic na hranách CFG nebo `LeadVector` pro uložení vektoru v množině generátorů, včetně informací o jeho vedoucím prvku a vedoucím indexu.

Samostatné třídy `LinearEquations` a `WriteProgram` se starají o konečný výpočet soustav lineárních rovnic v jednotlivých uzlech a konečný výpis analyzovaného programu do nového souboru, včetně vypsání lineárních vztahů, které v jednotlivých bodech programu platí.



Obrázek 7 - Zjednodušený třídní diagram části aplikace pro provedení interprocedurální analýzy

5.2 Popis tříd

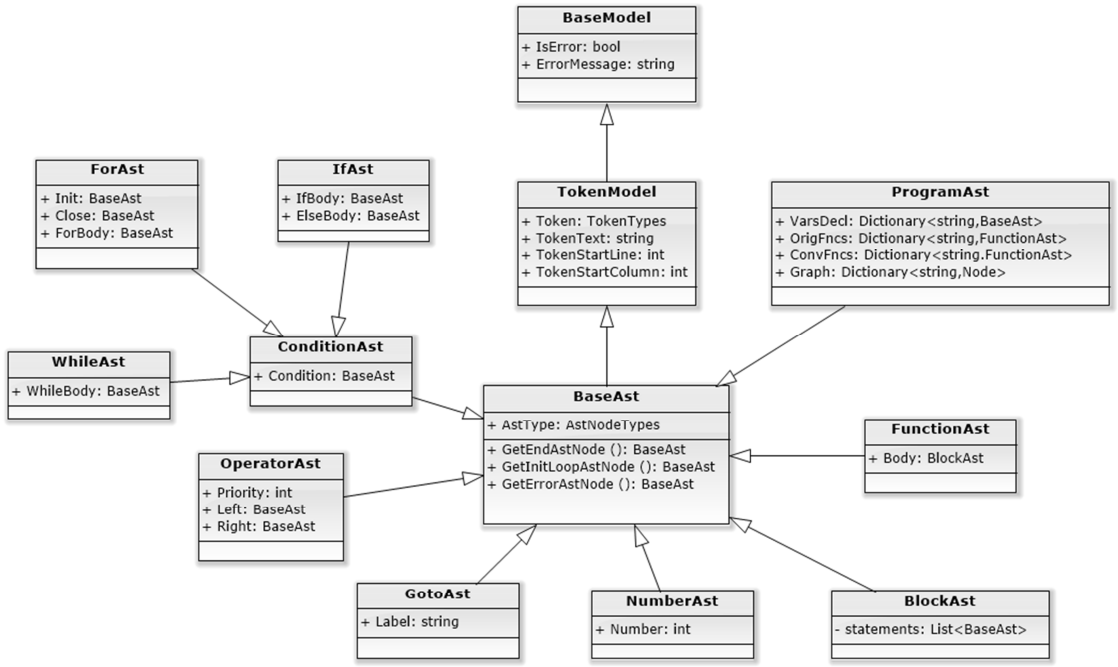
Implementace byla provedena dle třídního diagramu uvedeného v návrhu aplikace (Obrázek 6 a Obrázek 7). Případné drobné odchylky jsou uvedeny dále v rámci popisu implementace jednotlivých tříd.

Třída Program

Třída obsahuje pouze metodu `main`. Zajišťuje načtení analyzovaného programu a řídí běh aplikace postupným voláním metod za účelem realizace jednotlivých dílčích úkonů. Je spojovací třídou mezi částí pro zpracování vstupního textového souboru a samotnou interprocedurální analýzou.

Třídy modelů

V souboru Models.cs jsou sdruženy dohromady třídy tvořící modely tokenů a uzlů AST a třídy sloužící pro uložení a zpracování CFG. Modely uzlů AST jsou sestaveny tak, aby v sobě každý uzel uchovával svůj vlastní podstrom. Dědičnost a atributy tříd definujících modely pro tokeny a uzly jsou zachyceny na třídním diagramu v návrhu části aplikace sloužící pro zpracování vstupního textového souboru. Obrázek 8 zachycuje výřez z třídního diagramu, týkající se pouze tříd definujících modely.



Obrázek 8 - Třídní diagram modelů pro tokeny a uzly

Z diagramu je zřejmé, že třídy definující jednotlivé uzly obsahují jako atributy další uzly. Například třída `WhileAst`, definující uzel příkazu „while“ obsahuje jako atribut instanci třídy `BaseAst` (což je třída definující obecný uzel, který si pouze nese informaci o svém typu jako atribut), která může obsahovat tělo příkazu „while“. Zároveň třída `WhileAst` dědí ze třídy `ConditionAst`, takže obsahuje i instanci třídy pro uložení podstromu s podmínkou příkazu „while“. Obdobně třída `OperatorAst` obsahuje atributy `Left` a `Right`, které jsou instancemi třídy `BaseAst`, čímž je zajištěno uložení podstromů na levé i pravé straně každého operátoru. Třída `OperatorAst` obsahuje navíc atribut `Priority`, který slouží ke správnému sestavení AST pro určení priority operátorů. Je tedy zajištěno, že každý uzel nese informaci o celém svém podstromu a postupným skládáním uzlů dohromady lze sestavit AST celého analyzovaného programu.

Třída LexicalAnalyzer

Třída zajišťuje provedení lexikální analýzy. Obsahuje jedinou veřejnou metodu `ReadNextToken`, která je volána syntaktickou analýzou, a která zajišťuje předání aktuálního a následujícího tokenu. Při prvním požadavku na předání tokenu je zavolána metoda `GetTokensFromFile`, která zajistí zpracování vstupního textu a jeho transformaci na tokeny. Pro identifikaci jednotlivých tokenů je určena metoda `GetToken`, která čte vstupní text po jednotlivých znacích a tvoří jednotlivé tokeny. Na základě aktuálně přečteného a případně následujícího znaku metoda rozhoduje o typech jednotlivých tokenů. Typy tokenů jsou definovány výčtovým typem `TokenTypes`. Metoda `GetToken` implementuje DKA, který je naznačen na Obrázku 2 v kapitole popisující lexikální analýzu. Pro identifikaci jednotlivých přečtených znaků slouží pomocné metody, které rozpoznávají písmena, číslice a případné speciální znaky.

Třída SyntacticAnalyzer

Třída `SyntacticAnalyzer` je volána pomocí metody `GetAST`. Pro vyžádání dalších tokenů z lexikální analýzy používá metodu `ReadNextAst`. Každý token je ihned po předání syntaktické analýze konvertován na příslušný typ uzlu AST pomocí metody `GetAstNode`. Samotnou konverzi na příslušný typ uzlu AST realizuje generická metoda `ConvertTo<T>` omezená pouze na datové typy třídy `BaseAst`. Podle typu aktuálně přečteného tokenu je volána buď metoda pro zpracování deklarace proměnných (`GetVariables`) nebo metoda pro zpracování funkce (`GetFunctionAST`).

Metoda `GetVariables` zajišťuje rozpoznání deklarace a případné inicializace proměnných a ukládá je do seznamu proměnných a jejich deklarací v instanci třídy `ProgramAst`. Správné sestavení výrazu, pokud je proměnné inicializovaná výrazem, zajišťuje metoda `GetExprAST`.

Metoda `GetFunctionAST` sestavuje AST pro každou funkci analyzovaného programu. Všechny funkce ukládá do seznamu funkcí v instanci třídy `ProgramAst`. Pro sestavení těla funkce používá metodu `GetFncBodyAST`, která pomocí metody `GetStatementAST` rozpoznává načtené tokeny a podle nich volá metody pro zpracování jednotlivých příkazů („if“, „for“, „while“, „goto“, „return“, atd.) a výrazů. Pro sestavení podmínky příkazu je použita metoda `GetCondAST`.

Metody `GetExprAST` a `GetCondAST` používají pro svou činnost metodu `GetSubExprAST`. Tato metoda slouží k sestavení výrazů. Podle načteného tokenu rozpoznává proměnné, čísla, operátory a vnořené výrazy a podmínky a podle předem definované priority operátorů z nich sestavuje AST daného výrazu. V případě vnořených výrazů a podmínek se metoda volá rekurzivně, úroveň zanoření je ošetřena pomocí parametru metody `level`.

Třída SemanticAnalyzer

Jediným úkolem třídy `SemanticAnalyzer` je provedení kontroly, zda jsou všechny volané funkce v načteném programu deklarovány. Za tímto účelem obsahuje metody sloužící k procházení AST vytvořeného syntaktickou analýzou. Důležitá je metoda `CheckFunctionCallsInStatementAST`, která rozpoznává jednotlivé typy příkazů a v návaznosti na to pak volá příslušné metody. Ty rekurzivně opět volají metodu `CheckFunctionCallsInStatementAST`, čímž dochází k postupnému procházení stromu do hloubky, dokud buď běh programu nedojde na konec aktuálně čteného podstromu, nebo není nalezen uzel typu `FunctionCall`. V případě nalezení konce podstromu se pokračuje dalším podstromem, v případě nalezení uzlu typu `FunctionCall` dochází k ověřené, zda je název volané funkce uveden v seznamu deklarovaných funkcí. Pokud ano, pokračuje se dalším prohledáváním stromu, v opačném případě je vyvolána chyba a běh programu se zastaví.

Třída StatementConverter

Třída `StatementConverter` slouží pro převod vytvořeného AST na posloupnost příkazů řízených pouze příkazy „if“ a „goto-label“ dle postupu popsaného v kapitole 3.5 Vytvoření CFG. Obdobným způsobem jako třída `SemanticAnalyzer` prochází AST, v metodě `ConvertStatement` rozlišuje typ načteného příkazu („if“, „while“, „for“) a podle toho pak volá příslušnou metodu. Pokud je typ `block` (tedy nejedná se o žádný konečný příkaz), pak se metoda volá rekurzivně, dokud neobdrží buď konečný příkaz, nebo jeden z výše uvedených. Výstupem zpracování kódu třídou `StatementConverter` je požadovaná posloupnost příkazů, uložená v seznamu konvertovaných funkcí v instanci třídy `ProgramAst`.

Třída GraphGenerator

Třída `GraphGenerator` obsahuje jedinou metodu, která čtením posloupnosti příkazů sestavené třídou `StatementConverter`, zajišťuje vytvoření jednotlivých CFG pro interprocedurální analýzu.

Třída BaseFunctions

Třída obsahuje základní metody pro výpočty matic a vektorů, které se používají při zpracování interprocedurální analýzy. Součástí této třídy jsou mimo jiné metody pro výpočet rozkladu dle vztahu (4.2). S ohledem na časté použití rozkladu dle vztahu (4.2), např. v ALG-II.1 nebo při výpočtu soustavy lineárních rovnic, je zřejmé, že zjištění hodnoty r se bude v průběhu zpracování programu často opakovat a je tedy třeba definovat rychlý způsob vyhodnocení tohoto rozkladu.

Jelikož r definované v kapitole 4.2 odpovídá v bitovém zápisu čísla počtu nul na konci tohoto zápisu, lze hodnotu 2^r získat jako bitový součet čísla x, jehož rozklad je hledán a čísla $-x$. Pro zjištění hodnoty r z čísla 2^r lze použít např. metodu `Math.Log(Double,Double)`, která je součástí knihovny .NET. Nicméně s ohledem na opakovaně prováděné výpočty je jednoznačně efektivnější provést výpočet jedenkrát, výsledek uložit a následně již pouze v případě potřeby přečíst výsledek. Za tímto účelem jsou tedy ve třídě `BaseFunctions` implementovány metody pro výpočet převodní tabulky `rArray`, která po vypočtení obsahuje hodnoty, ze kterých lze jednoduše přečíst hodnotu r pro každé 2^r.

Jestliže výpočty jsou řešeny pro 2^w, pak je pro výpočet převodní tabulky třeba určit nejprve nejbližší prvočíslo větší než w. Následně je tabulka sestavena tak, že je pro každé r vypočten index, na jehož pozici je uložena hodnota r. Hledaný index je vypočten dle vzorce $index = 2^r \,\%\, prvočíslo$. V průběhu ukládání hodnot do tabulky je kontrolováno, zda nedochází k uložení hodnoty r na index, který již byl použit pro jiné r. Pokud ano, nalezne se další vyšší prvočíslo a postup se opakuje od začátku, dokud nejsou uloženy všechny hodnoty r až do w. Na závěr výpočtu tabulky je třeba na index 0 uložit hodnotu w jako výjimku, neboť platí, že $2^w = 0$, což však bez uplatnění této výjimky nebude v tabulce zachyceno. Kdykoli v průběhu zpracování programu je pak třeba vypočíst hodnotu r ze zjištěné hodnoty 2^r, stačí provést výpočet indexu a přečíst hodnotu r z tabulky. Např. pro $w = 8$, tak bude tabulka `rArray` vypadat takto:

index	0	1	2	3	4	5	6	7	8	9	10
r	8	0	1	0	2	4	0	7	3	6	5

Tabulka 4 - převodní tabulka `rArray`

Po vypočtení hodnoty r stačí pro stanovení hodnoty d, tedy lichého čísla z rozkládaného čísla x, provést pouze pravý bitový posun čísla x o hodnotu r. Metoda pro výpočet rozkladu dle vztahu (4.2) je uvedena v ukázce programu na Obrázku 9.

Třída kromě informace o hodnotě w a r obsahuje také informace o hodnotě modula, ve kterém výpočty probíhají a o počtu proměnných analyzovaného programu. Hodnota w je parametr, který se předává aplikaci při jejím spuštění a je limitována hodnotami $1 \leq w < 64$. Otevřenost intervalu na pravé straně je dána nutností použít při implementaci algoritmu datový typ `long`, který je znaménkový, a tudíž maximální velikost modula může být 2^{63}. Aby bylo možno povolit tento interval uzavřený (čímž by bylo umožněno povolit výpočty modulo 2^{64}), bylo by třeba použít neznaménkový datový typ `ulong`. Tím by však vznikly problémy při výpočtech, neboť vypočítávané hodnoty v některých případech mohou nabývat i záporných hodnot.

```
public int Reduction(long nr, out long d)
{
    if ((nr % 2) != 0)
    {
        d = nr;
        return 0;
    }

    int r = r_arr[(nr & (-nr)) % prime];
    d = (nr >> r);
    return r;
}
```

Obrázek 9 - Metoda pro výpočet rozkladu dle vztahu (4.2)

Třída InterproceduralAnalyzer

Třída `InterproceduralAnalyzer` implementuje algoritmy popsané v kapitole 4.3 a v příloze II. Zajišťuje sestavení změnových matic na hranách funkcí, výpočet množin změnových matic jednotlivých funkcí a jejich distribuci na hrany, které tyto funkce volají, výpočet množin generátorů v uzlech a závěrečný průchod grafy pro konečný výpočet lineárních vztahů.

Třída implementuje fronty vyžadované algoritmy pro výpočet analýzy. Pro ukládání dvojic do front používá třídy `NodeMatrix` při výpočtu změnových matic funkcí a `NodeVector` při výpočtu množin generátorů. Dále pro svou činnost používá třídy `TransitionMatrixSet`, `GeneratorSet` a `LeadVector` popsané níže.

Třída TransitionMatrixSet

Třída `TransitionMatrixSet` slouží pro uložení množin změnových matic na hranách CFG a obsahuje metody pro sestavení těchto matic. Pro sestavení změnové matice z výrazu je vždy použita jednotková matice, ve které se vynuluje jednička v řádku odpovídajícím proměnné, která je daným výrazem měněna. Následně dochází k průchodu a vyhodnocení AST obsahujícího zkoumaný výraz a podle hodnot uvedených u jednotlivých proměnných k dosazení příslušných čísel do sestavované matice. Pokud je výraz složen z binárních i unárních operátorů (např. $x = 1 + 2x + (y + +) + z$), je změnová matice sestavena pomocí přetížené metody `ProceedUnary`, která složený výraz rozloží a sestaví příslušnou změnovou matici.

Princip sestavení změnové matice pro složený výraz spočívá v tom, že v případě, že má být proměnná zvýšena (resp. snížena) unárním operátorem až po vyhodnocení výrazu, pak se pouze v matici na řádek příslušející k proměnné s unárním operátorem, přičte (resp. odečte) 1 v prvním sloupci. Pokud má být zvýšení (resp. snížení) proměnné unárním operátorem provedeno ještě před vyhodnocením výrazu, pak se kromě postupu popsaného v předchozí větě ještě přičte 1 ke konstantě v prvním sloupci na řádku proměnné, která je upravována vyhodnocovaným výrazem.

Složitější výrazy vyžadující např. roznásobení závorky apod. jsou vyhodnoceny jako $x = ?$ a na hranu jsou pak uloženy matice pro $x = 0$ a $x = 1$, x obecně značí příslušnou proměnnou, která je výrazem upravovaná.

Třída GeneratorSet

Tato třída slouží pro ukládání množin generátorů v uzlech CFG. Je použita jak pro dočasné ukládání množin generátorů při výpočtu změnových matic funkcí, tak i později pro ukládání množin generátorů, ze kterých se vypočítávají konečné vztahy mezi proměnnými. Třída

obsahuje metodu `AddVector`, která implementuje algoritmus ALG-II.1 a k ní pomocné metody `AddEven` pro kontrolu, zda zůstává množina saturovaná při přidávání vektoru se sudým vedoucím prvkem, `InsertVector` pro zachování množiny generátorů ve schodovitém tvaru při přidávání vektoru a `RemoveVector` pro úpravu indexů vektorů v množině generátorů při odebrání vektoru.

Třída LeadVector

Třída slouží k uložení vektoru z množiny generátorů. Kromě samotného vektoru obsahuje i informaci o vedoucím prvku a vedoucím indexu daného vektoru.

Třída LinearEquations

Třída `LinearEquations` implementuje sestavení matic z množin generátorů a jejich konečný výpočet jako soustavy lineárních rovnic. Zajišťuje tedy výpočet diagonální matice, sestavení matice R^{-1} a výpočet konečného řešení zachycujícího lineární vztahy mezi proměnnými vstupního analyzovaného programu.

Třída WriteProgram

Tato třída slouží pro vypsání původního programu zbaveného všech původních komentářů a doplněného o vztahy mezi proměnnými zachycenými v afinních relacích. Tyto vztahy jsou do nově vypsaného původního programu vypsány formou komentářů.

6 Testování aplikace

Aplikace `InterproceduralAnalysis` je konzolová aplikace a spouští se pomocí souboru intprocan.exe. Příkaz ke spuštění aplikace obsahuje několik povinných i nepovinných parametrů. Kompletní formát příkazu ke spuštění aplikace vypadá takto:

```
intprocan.exe /p <nazev_programu> /w <rozsah> /d [all+la+sa+iam+iag+iale]
```

Volitelné parametry na konci příkazu ovlivňují, zda budou v průběhu zpracování analyzovaného programu vypisovány na příkazovou řádku průběžné výpisy aplikace, či nikoli. Lze vybrat z těchto možností:

- `all` - všechny průběžné výpisy
- `la` - výpis z průběhu lexikální analýzy
- `sa` - výpis z průběhu syntaktické analýzy
- `iam` - výpis zpracování algoritmu interprocedurální analýzy při výpočtu změnových matic
- `iag` - výpis zpracování algoritmu interprocedurální analýzy při výpočtu množin generátorů
- `iale` - výpis zpracování algoritmu interprocedurální analýzy při výpočtu konečných hodnot lineárních vztahů (výpočet soustav lineárních rovnic)

Parametry `<nazev_programu>` a `<rozsah>` jsou povinné. Pokud je soubor s analyzovaným programem uložen ve stejném adresáři jako samotná aplikace, stačí v tomto parametru uvést pouze název souboru s analyzovaným programem, v opačném případě je třeba uvést celou cestu k souboru. Rozsah slouží k určení modula, v jakém má výpočet probíhat. Zadává se hodnota w, která určuje mocnitele 2 ze vztahu $modulo = 2^w$.

Po spuštění aplikace je načten text ze vstupního souboru a pomocí příslušných tříd zkontrolována formální správnost vstupního textu. Následně je zpracován obsah vstupního textu do strukturované podoby AST a dále prostřednictvím převodu na posloupnost příkazů jsou vytvořeny CFG pro jednotlivé funkce vstupního programu.

Vytvořené CFG jsou pak procházeny jednotlivými algoritmy pro zpracování interprocedurální analýzy. Po dokončení všech výpočtů jsou v uzlech CFG uloženy vektory obsahující hodnoty afinních relací, vyjadřujících vztahy mezi proměnnými v daném bodu programu. Výstupem aplikace je nově vytvořený soubor se stejným názvem jako měl vstupní soubor rozšířeným o příponu „.ia“, který obsahuje přepis původního programu s odstraněnými komentáři a doplněný o vypočtené afinní relace za každým příkazem programu. Vypočtené afinní relace jsou do textu vloženy formou komentářů. Výstupní soubor má stejný formát jako původní soubor a je uložen ve stejném adresáři jako analyzovaný soubor.

Testování aplikace je provedeno na několika programech. Nejprve je nutné ověřit správnost kompletního zpracování na jednoduchých programech. Následně je třeba ověřit funkčnost aplikace na několika triviálních programech, na kterých lze zpětně provést ověření platnosti vypočtených vztahů. Na závěr je provedeno ověření časové složitosti algoritmu, resp. nárůst času zpracování v návaznosti na množství proměnných analyzovaného programu.

6.1 Ověření správnosti zpracování

Testování správnosti zpracování vstupního programu bylo provedeno na dvou programech, z nichž jeden má pouze funkci `main` a druhý je složen ze dvou funkcí, z nichž jedna je funkce

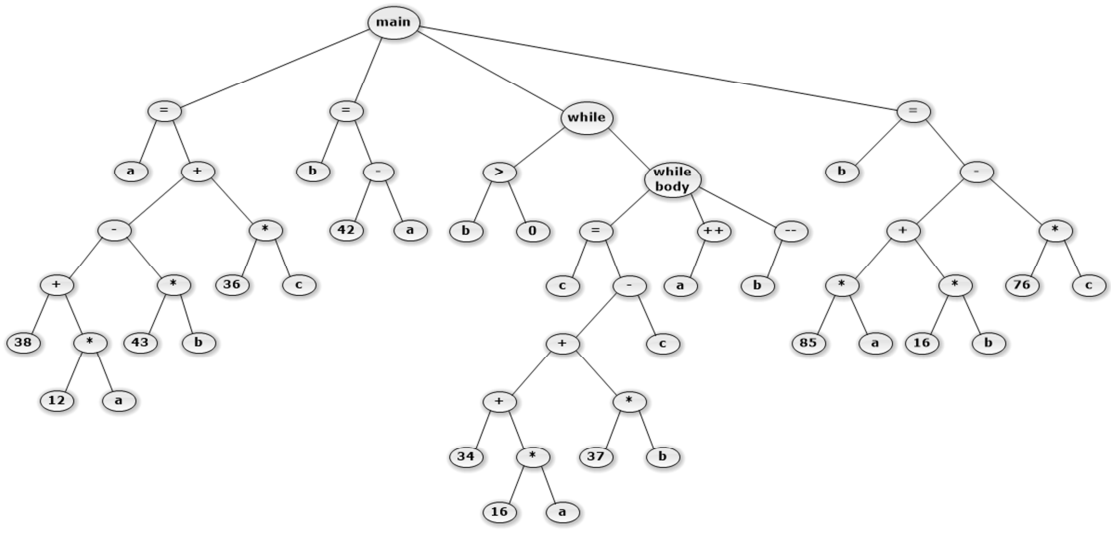
`main`. Pro tento typ testu byly zvoleny složitější konstrukce programů kvůli ověření správnosti sestavení AST, posloupnosti příkazů a CFG.

6.1.1 Test jedné funkce

Test byl proveden na programu uvedeném na Obrázku 10. Očekává se sestavení AST dle Obrázku 11. Pro sestavení posloupnosti příkazů je očekávaný výstup uveden v Tabulce 5 v levém sloupci. V pravém sloupci je zároveň připojen výpis sestavené posloupnosti příkazů, který aplikace vypíše na vyžádání na příkazový řádek. Na Obrázku 12 je pak připojen CFG sestavený dle posloupnosti příkazů.

```
var a,b,c;

function main()
{
  a = 38 + 12*a - 43*b + 36*c;
  b = 42 - a;
  while ( b > 0 )
  {
    c = 34 + 16*a + 37*b - c;
    a++;
    b--;
  }
  b = 85*a + 16*b - 76*c;
}
```

Obrázek 10 - Testovací program pro test 6.1.1

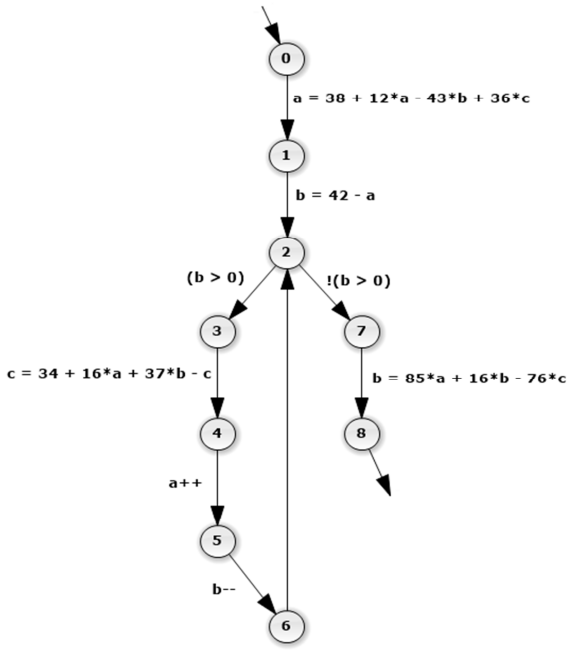


Obrázek 11 - AST pro testovací program 6.1.1

Manuální sestavení	Výstup aplikace

MAIN:
1: a = 38 + 12*a - 43*b + 36*c
2: b = 42 - a
3: if ! (b > 0) goto 8
4: c = 34 + 16*a + 37*b - c
5: a++
6: b--
7: goto 3
8: b = 85*a + 16*b - 76*c
	function main()
	{
	a = (((38 + (12 * a)) - (43 * b))
	+ (36 * c));
	b = (42 - a);
	$WhileBegin1:
	if (!((b > 0)))
	goto $WhileEnd1;
	$WhileTrue1:
	c = (((34 + (16 * a)) + (37 * b))
	- c);
	a++;
	b--;
	goto $WhileBegin1;
	$WhileEnd1:
	b = (((85 * a) + (16 * b)) - (76 *
	c));
	}

Tabulka 5 - Posloupnost příkazů pro testovací program 6.1.1



Obrázek 12 - CFG pro testovací program 6.1.1

Jelikož aplikace neprovádí grafický výpis AST ani CFG, bylo sestavení těchto grafů ověřeno vizuálně v ladícím módu aplikace. Porovnání sestavené posloupnosti příkazů a vstupu aplikace je k dispozici v Tabulce 5 a je zřejmé, že obě varianty jsou významově shodné.

6.1.2 Test více funkcí

Obdobným způsobem, jako u předchozího testu (v Kapitole 6.1.1) byl proveden i test pro více funkcí. Program, sestavené AST a CFG i tabulka s porovnáním sestavené posloupnosti příkazů, jsou uvedeny v příloze č. IV.1. I na více funkcích byla ověřena správná funkčnost.

6.2 Ověření správnosti výpočtů

Ověření správnosti výpočtů bylo provedeno na triviálních programech, na kterých lze snadno ověřit platnost vypočtených vztahů. U všech testů byla použita hodnota $w = 3$ z důvodu jednodušších výpočtů.

První ověření bylo provedeno na programu uvedeném v Tabulce 6, kde je zároveň uveden i obsah souboru, který je výstupem zpracování analýzy vstupního programu.

Analyzovaný program	Výstup

`var a, b;`
 	
`function main()`
`{`
`  a = 0;`
`  if (a > b)`
`  {`
`  a = 1;`
`  }`
`}`
	`var a, b;`
	
	`function main()`
	`  /*`
	`  * Zadna omezeni`
	`  */`
	`{`
	`  a = 0;`
	`  /*`
	`  * 1*a = 0`
	`  */`
	`  if (a > b)`
	`  /*`
	`  * 1*a = 0`
	`  */`
	`  {`
	`  a = 1;`
	`  /*`
	`  * 1*a = 1`
	`  */`
	`  }`
	`  /*`
	`  * Zadna omezeni`
	`  */`
	`}`

Tabulka 6 - Testovací program 6.2.1

Z výstupu uvedené v Tabulce 6 je patrné, že na vstupu funkce `main` nejsou proměnné ničím omezeny. S ohledem na skutečnost, že nejsou při deklaraci inicializované, je tento závěr pravdivý. Za první příkazem `a = 0` dochází k omezení proměnné `a` výrazem `1*a = 0`. Jelikož jsou výsledky výpočtů soustav lineárních rovnic zachyceny v množinách generátorů, znamená to, že i lineární vztahy vypsané ve výstupním souboru jsou pouze zástupci všech vztahů, které v daném bodě programu platí. Všechna řešení, která v daném bodě platí, jsou pak lineární kombinací vypsaných vztahů. Proto výraz `1*a = 0` ve skutečnosti znamená, že proměnná `a` musí být vždy násobena nějakou konstantou, aby daný vztah platil. Jelikož je vidět, že tento

vztah je uveden za příkazem, který proměnné `a` přiřazuje hodnotu nula, pak je zřejmé, že je tento vztah platný.

Podmínka větvení, jako další řádek programu, neobsahuje žádné příkazy, které by ovlivňovaly hodnoty proměnných, proto i vztah vypsaný za touto podmínkou, je shodný se vztahem v předchozím kroku.

Další příkaz, `a = 1`, mění hodnotu přiřazenou do proměnné `a`, a proto musí dojít i ke změně výrazu vyjadřujícího vztah platný v tomto bodu programu. Pokud je do tohoto výrazu dosazena aktuální hodnota proměnné `a`, pak je zřejmé, že vypsaný vztah je opět platný, neboť $1 * 1 = 1$.

Analyzovaný program	Výstup

`var a, b;`
 	
`function main()`
`{`
`  a = 2;`
`  while (a > b)`
`  {`
`  b = 3;`
`  }`
`}`
	`var a, b;`
	
	`function main()`
	`  /*`
	`  * Zadna omezeni`
	`  */`
	`{`
	`  a = 2;`
	`  /*`
	`  * 1*a = 2`
	`  * 4*a = 0`
	`  */`
	`  while (a > b)`
	`  /*`
	`  * 1*a = 2`
	`  * 4*a = 0`
	`  */`
	`  {`
	`  b = 3;`
	`  /*`
	`  * 1*b = 3`
	`  * 5*a + 2*b = 0`
	`  */`
	`  }`
	`  /*`
	`  * 1*a = 2`
	`  * 4*a = 0`
	`  */`
	`}`

Tabulka 7 - Testovací program 6.2.2

Druhý test je zachycen v Tabulce 7. Opět díky neinicializovaným proměnným není na vstupu funkce `main` žádné omezení a stejně tak lze prostým dosazením ověřit, že vztahy vypočtené v jednotlivých bodech programu jsou platné. Na první pohled se však zdá zarážející skutečnost, že vztah za blokem `while` není tímto cyklem ovlivněn. Pokud je však důkladně prozkoumán průběh algoritmu, lze poměrně snadno zjistit, že v průběhu zpracování algoritmu pro výpočet množiny generátorů jsou na vstupu pro výpočet množiny pro uzel za příkazem `b = 3` dva vektory a vyhodnocením tohoto výrazu zůstane pouze jediný vektor pro uzel odpovídající bodu programu na konci bloku `while`. Následným spuštěním algoritmu ALG-II.1 pro přidání tohoto vektoru do uzlu odpovídajícímu bodu programu za blokem `while`, dochází

k redukci tohoto vektoru. Z tohoto postupu tedy vyplývá, že obsah cyklu `while` omezení pro proměnné dále nerozšíří.

Analyzovaný program	Výstup

`var a,b,c;`
 	
`function main()`
`{`
`  a = 3;`
`  fncA();`
`  b = a + b;`
`}`
 	
`function fncA()`
`{`
`  b = 4;`
`  c = a + b + 2;`
`}`
	`var a, b, c;`
	
	`function main()`
	`  /*`
	`   * Zadna omezeni`
	`  */`
	`{`
	`  a = 3;`
	`    /*`
	`     * 1*a = 3`
	`    */`
	`  fncA();`
	`    /*`
	`     * 1*a = 3`
	`     * 1*a + 5*c = 0`
	`     * 5*b + 4*c = 0`
	`    */`
	`  b = a + b;`
	`    /*`
	`     * 1*a = 3`
	`     * 7*a + 5*b = 0`
	`     * 3*b + 3*c = 0`
	`    */`
	`}`
	
	`function fncA()`
	`  /*`
	`   * 1*a = 3`
	`  */`
	`{`
	`  b = 4;`
	`    /*`
	`     * 1*a = 3`
	`     * 4*a + 5*b = 0`
	`     * 2*b = 0`
	`    */`
	`  c = a + b + 2;`
	`    /*`
	`     * 1*a = 3`
	`     * 1*a + 5*c = 0`
	`     * 5*b + 4*c = 0`
	`    */`
	`}`

Tabulka 8 - Testovací program 6.2.3

Třetí test (Tabulka 8) ověřuje platnost výstupů při volání funkcí. Platnost vypočtených vztahů lze opět ověřit prostým dosazením aktuálních hodnot proměnných do jednotlivých vztahů. Pro názornou kontrolu lze provést ověření vztahů platných na konci funkce `main` (tedy na konci zpracování programu):

Hodnoty proměnných po skončení vykonávaní programu:

$a = 3$

$b = a + b = 3 + 4 = 7$

$c = a + b + 2 = 3 + 4 + 2 = 9 \equiv 1\ (mod\ 8)$

Ověření platnosti vztahů vypočtených pro uzel odpovídající konci programu (tj. konci funkce `main`):

$1 * a = 1 * 3 = 3$

$7 * a + 5 * b = 7 * 3 + 5 * 7 = 56 \equiv 0\ (mod\ 8)$

$3 * b + 3 * c = 3 * 7 + 3 * 1 = 24 \equiv 0 (mod\ 8)$

Analyzovaný program	Výstup

`var a = 2;`
`var b = 3;`
`var c = 4;`
 	
`function main()`
`{`
`  a = b + c;`
`  fncA();`
`}`
 	
`function fncA()`
`{`
`  b = 6 + 3*a + 4*c;`
`  fncA();`
`  a = 3*b + c;`
`}`
	`var a = 2;`
	`  /*`
	`   * 1*a = 2`
	`   * 4*a = 0`
	`  */`
	`var b = 3;`
	`  /*`
	`   * 1*b = 3`
	`   * 5*a + 2*b = 0`
	`  */`
	`var c = 4;`
	`  /*`
	`   * 1*b = 3`
	`   * 5*a + 2*b = 0`
	`   * 4*b + 5*c = 0`
	`   * 2*c = 0`
	`  */`
	
	`function main()`
	`  /*`
	`   * 1*b = 3`
	`   * 5*a + 2*b = 0`
	`   * 4*b + 5*c = 0`
	`   * 2*c = 0`
	`  */`
	`{`
	`  a = b + c;`
	`    /*`
	`     * 1*a = 7`
	`     * 3*a + 1*b = 0`
	`     * 4*b + 3*c = 0`
	`     * 6*c = 0`
	`    */`
	`  fncA();`
	`    /*`
	`     * Nedosazitelny stav`
	`    */`
	`}`
	
	`function fncA()`
	`  /*`
	`   * 1*a = 7`

```
   * 3*a + 1*b = 0
   * 4*b + 3*c = 0
   * 6*c = 0
  */
{
  b = 6 + 3 * a + 4 * c;
    /*
     * 1*a = 7
     * 3*a + 1*b = 0
     * 4*b + 3*c = 0
     * 6*c = 0
    */
  fncA();
    /*
     * Nedosazitelny stav
    */
  a = 3 * b + c;
    /*
     * Nedosazitelny stav
    */
}
```

Tabulka 9 - Testovací program 6.2.4

Čtvrtý test (Tabulka 9) prokazuje správné odhalení nedosažitelných stavů při cyklickém volání funkcí. Ve funkci `main` se nedosažitelný stav objevuje po volání funkce `fncA`. Důvodem je, že funkce `fncA()` se pokouší o rekurzivní volání, nicméně díky tomu skončí v nekonečné smyčce. Stavy, které jsou ve funkci `fncA` za tímto rekurzivním voláním jsou tedy také nedosažitelné. Obdobně dochází k vyhodnocení nedosažitelných stavů, pokud program obsahuje funkci (kromě funkce `main`), která není volána z jiné funkce.

6.3 Faktory ovlivňující rychlost zpracování

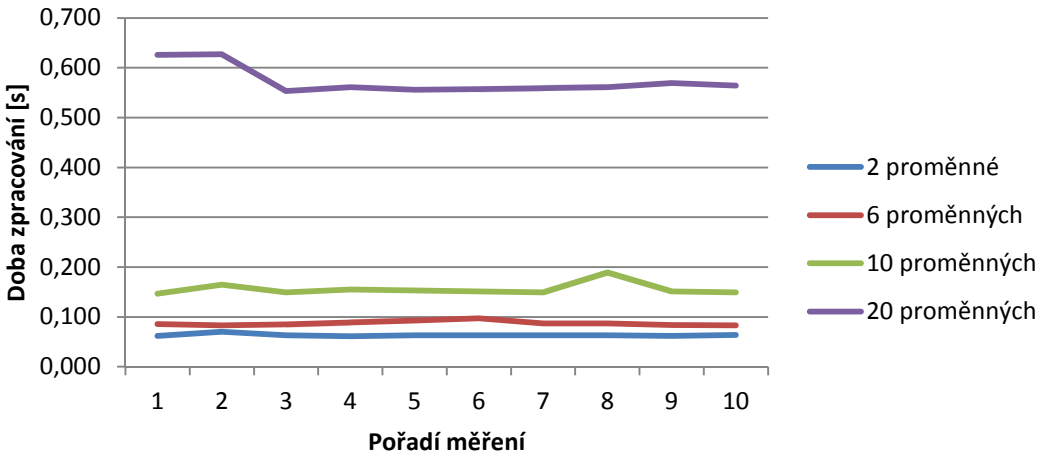
Počet proměnných	Naměřený čas zpracování [s] pro w = 8										Průměrný čas zpracování [s]
	1	2	3	4	5	6	7	8	9	10	
2	0,062	0,070	0,063	0,061	0,063	0,063	0,063	0,063	0,062	0,064	0,063
3	0,064	0,079	0,063	0,065	0,070	0,064	0,064	0,065	0,064	0,071	0,067
4	0,068	0,069	0,070	0,073	0,068	0,073	0,069	0,068	0,068	0,068	0,069
5	0,074	0,074	0,073	0,087	0,080	0,089	0,079	0,073	0,075	0,089	0,079
6	0,086	0,083	0,085	0,089	0,093	0,097	0,087	0,087	0,084	0,083	0,087
7	0,095	0,096	0,097	0,096	0,108	0,109	0,110	0,099	0,097	0,101	0,101
8	0,114	0,131	0,129	0,125	0,109	0,112	0,112	0,109	0,110	0,110	0,116
9	0,132	0,139	0,144	0,152	0,136	0,143	0,136	0,131	0,129	0,134	0,138
10	0,147	0,165	0,149	0,155	0,153	0,151	0,149	0,189	0,151	0,149	0,156
20	0,626	0,627	0,553	0,561	0,556	0,557	0,559	0,561	0,569	0,564	0,573

Tabulka 10 - Porovnání vlivu počtu proměnných na dobu zpracování

Pro ověření vlivu některých parametrů na rychlost zpracování algoritmu byl použit program uvedený v příloze IV.2. Testován byl vliv počtu proměnných, velikost zvoleného modula a složitost analyzovaného programu. Program, který byl použit pro testování vlivu počtu proměnných a velikosti zvoleného modula, je v příloze uveden ve své nejdelší variantě s 20 proměnnými. Pro testování byl počet proměnných postupně od konce mazán.

Čas zpracování nebyl ovlivňován délkou analyzovaného programu ve smyslu počtu znaků programu, protože měření času bylo uplatněno pouze na zpracování samotné interprocedurální analýzy, nebylo do něj zahrnuto ani předzpracování analyzovaného programu před zahájením samotné interprocedurální analýzy, ani následný výpis výsledků do nového souboru.

Testování bylo provedeno na počítači s operačním systémem Windows 8.1 (64bit), s procesorem Intel Core i7-3517U CPU 1,9 GHz, 8GB RAM.



Obrázek 13 - Graf porovnání vlivu počtu proměnných na dobu zpracování

V Tabulce 10 jsou uvedeny průměrné hodnoty získané z 10 po sobě jdoucích měření. Zvyšující se výpočetní náročnost je zřejmá nejen z postupně se zvyšujícího času zpracování, ale i z rozdílů časů po sobě jdoucích měření (Tabulka 11). Z grafu na Obrázku 13 je patrné, že zatímco ještě při 6 proměnných je doba výpočtu algoritmů téměř konstantní, při 10 a 20 proměnných se již projevují výkyvy v naměřeném čase.

Počet proměnných	Minimální naměřený čas	Maximální naměřený čas	Rozdíl
2	0,070	0,061	0,009
3	0,079	0,063	0,016
4	0,073	0,068	0,005
5	0,089	0,073	0,016
6	0,097	0,083	0,014
7	0,110	0,095	0,015
8	0,131	0,109	0,022
9	0,152	0,129	0,023
10	0,189	0,147	0,042
20	0,627	0,553	0,074

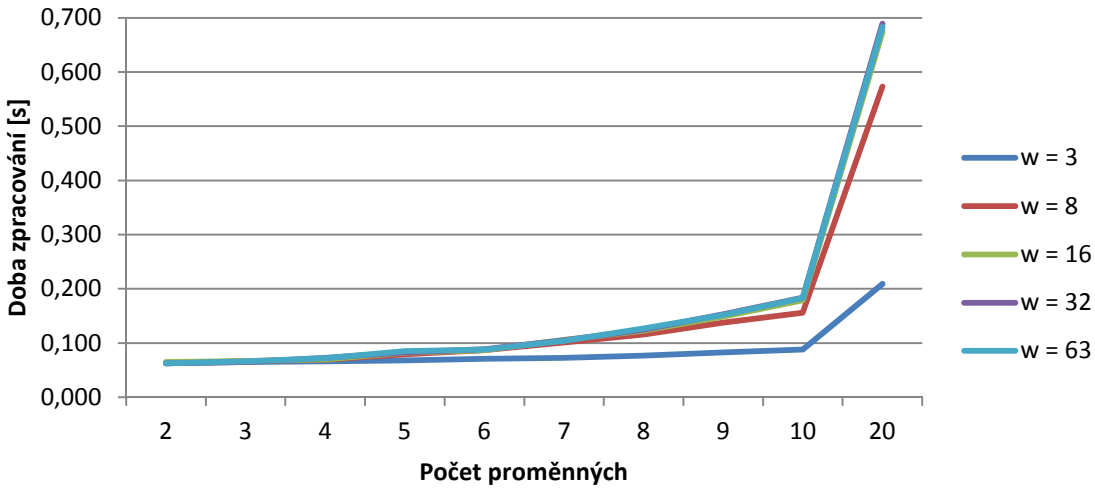
Tabulka 11 - Porovnání rozdílu v naměřených hodnotách z Tabulky 10

Prokázaná závislost rychlosti zpracování na počtu proměnných byla očekávaným jevem s ohledem na skutečnost, že algoritmy pracují s maticemi a vektory, jejichž rozměry jsou dány právě počtem proměnných. V případě zde testovaných 20 proměnných tak v aplikaci probíhají výpočty s maticemi o rozměrech 21 x 21 a s vektory o rozměrech 441 (při výpočtu změnových matic funkcí) a 21.

Počet proměnných	**Průměrný čas zpracování [s]**				
	w = 3	**w = 8**	**w = 16**	**w = 32**	**w = 63**
2	0,063	0,063	0,065	0,063	0,063
3	0,065	0,067	0,067	0,065	0,066
4	0,066	0,069	0,070	0,073	0,072
5	0,068	0,079	0,082	0,083	0,085
6	0,071	0,087	0,087	0,089	0,088
7	0,073	0,101	0,106	0,105	0,104
8	0,077	0,116	0,123	0,125	0,127
9	0,083	0,138	0,149	0,153	0,152
10	0,088	0,156	0,179	0,184	0,183
20	0,209	0,573	0,675	0,689	0,683

Tabulka 12 – Porovnání vlivu velikosti modula na dobu zpracování

Dále bylo testováno ovlivnění doby zpracování velikostí zadaného modula (Tabulka 12). Z výsledků je patrné, že velikost modula neovlivňuje rychlost zpracování algoritmu při nízkém počtu proměnných. Již při 5 proměnných se začíná projevovat výraznější rozdíl v době zpracování pro w = 3 od zbývajících testovaných hodnot. Při 8 proměnných se začíná objevovat výraznější rozdíl i pro w = 8. Při 20 proměnných lze pozorovat, že se začínají mírně lišit naměřené časy pro w = 16. Z výsledků je tedy zřejmé, že velikost modula má na rychlost zpracování algoritmů vliv ve vazbě na množství proměnných analyzovaného programu, nicméně se jedná o vliv poměrně malý (Obrázek 14).



Obrázek 14 - Graf vlivu velikosti modula a počtu proměnných na dobu zpracování

Uvedené závěry se netýkají výrazně malých programů, které jsou počtem proměnných i zvolenou velikostí modula ovlivněny jen minimálně, jak dokazuje měření na programu uvedeném v příloze č. IV.3. Měření je zachyceno v Tabulce 13.

Počet proměnných	**Naměřený čas zpracování [s] pro w = 8**										**Průměrný čas zpracování [s]**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	
2	0,058	0,058	0,061	0,065	0,059	0,058	0,059	0,059	0,059	0,059	0,060
3	0,058	0,059	0,060	0,059	0,071	0,059	0,068	0,058	0,059	0,058	0,061
4	0,058	0,059	0,060	0,059	0,060	0,062	0,059	0,059	0,060	0,060	0,060
5	0,058	0,063	0,058	0,075	0,074	0,077	0,063	0,062	0,075	0,077	0,068
20	0,067	0,072	0,067	0,067	0,066	0,067	0,068	0,066	0,066	0,066	0,067

Tabulka 13 - Porovnání vlivu počtu proměnných na dobu zpracování malého programu

7 Závěr

Cílem této diplomové práce bylo implementovat algoritmy navržené autory [2] a otestovat je na vhodných příkladech. Postupně bylo popsáno navržení bezkontextové gramatiky pro analyzovaný program, předzpracování analyzovaného programu lexikální, syntaktickou a sémantickou analýzou a vytvoření CFG, který může být zpracován implementovanými algoritmy. Dále byla popsána pravidla pro výpočty v modulární aritmetice se zaměřením na specifický případ modulo 2^w, postup sestavení matic z jednotlivých příkazů analyzovaného programu a princip výpočtů s maticemi a vektory. Následně byl vysvětlen postup, jakým navržené algoritmy zpracovávají analyzovaný program, popsána implementace programu, který realizuje navržené algoritmy a nakonec bylo popsáno ověření funkčnosti programu testováním.

Navržené algoritmy tedy byly implementovány jako součást aplikace, která jako svůj vstup načte soubor obsahující analyzovaný program. Program musí být zapsán v jazyce odpovídajícím navržené bezkontextové gramatice, což je aplikací při předzpracování ověřeno. Načtená data jsou následně zpracována do CFG odpovídajících jednotlivým funkcím analyzovaného programu. CFG jsou pak procházeny pomocí implementovaných algoritmů pro zjištění vztahů mezi hodnotami proměnných. Tyto vztahy jsou zachyceny v podobě lineárních výrazů. Výstupem aplikace je nový soubor, obsahující původní program doplněný o výpis lineárních výrazů zachycujících vztahy mezi hodnotami proměnných za každým příkazem v programu. Výpis vztahů je do původního programu proveden pomocí komentářů.

Bylo ověřeno, že implementovaný program provádí zpracování analyzovaného programu korektně a že výpočty prováděné navrženými algoritmy vracejí relevantní výsledky. V rámci testování bylo také dokázáno, že rychlost zpracování algoritmů je závislá především na počtu proměnných a v návaznosti na počet proměnných i na velikosti zvoleného modula.

Závěrem lze konstatovat, že se podařilo implementovat navržené algoritmy, jejichž funkčnost byla ověřena na testovacích programech a že aplikace pracuje dle požadavků. Je však také třeba konstatovat, že funkčnost aplikace je omezena jednak navrženou bezkontextovou gramatikou a dále také schopností vyhodnocovat výrazy vstupního programu, kdy složitější výrazy, vyžadující například roznásobení závorky, již vyhodnotí jako výrazy nesplňující požadavky na formát afinního přiřazení a podle toho s nimi pracuje.

Při implementaci algoritmů bylo třeba vyřešit několik problémů, které v rámci návrhu algoritmů v článku [2] nebyly řešeny, jako například algoritmy pro sestavení změnových matic z výrazů analyzovaného programu, nebo převedení AST vytvořeného syntaktickou analýzou na CFG, který bylo možné zpracovat implementovanými algoritmy. Všechny tyto postupy jsou v současnosti implementovány pouze pro bezkontextovou gramatiku navrženou pro potřeby této práce. Z praktického hlediska pro možnost budoucího využití těchto algoritmů v praxi by bylo možné stávající implementaci rozšířit pro nějaký běžně používaný programovací jazyk. Toto rozšíření by vyžadovalo novou implementaci předzpracování analyzovaného programu až do podoby CFG. Pro uložení CFG by již bylo možno využít implementované datové struktury, které mohou být zpracovány pomocí implementovaných algoritmů. S ohledem na vazbu na individuálně realizovanou lexikální a syntaktickou analýzu by bylo v závěru nutné ještě upravit třídu zajišťující závěrečný výpis vypočtených lineárních vztahů spolu s původním programem do nového souboru.

8 Literatura

[1] HABALA, Petr. *Diskrétní matematika: 7. Počítání modulo*. ČVUT, Fakulta elektrotechnická. Praha, 2012. Dostupné z: http://math.feld.cvut.cz/habala/teaching/dma/dmknih07.pdf

[2] MÜLLER-OLM, Markus a Helmut SEIDL. Analysis Of Mudular Arithmetic. In: *ACM Transactions on Programming Languages and Systems*. Special Issue ESOP'05, Volume 29, Issue 5. New York: Association for Computing Machinery, August 2007, 29:1-29:27. ISSN 0164-0925. DOI: 10.1145/1275497.1275504.

[3] SCHWARTZBACH, Michael I. *Lecture Notes on Static Analysis*. BRICS, Department of Computer Science, University of Aarhus. Denmark, 2006.

[4] Syntaktický strom. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-02-12]. Dostupné z: http://cs.wikipedia.org/wiki/Syntaktický_strom

[5] Zbytkové třídy. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-02-02]. Dostupné z: http://cs.wikipedia.org/wiki/Množina_zbytkových_tříd

9 Přílohy

I. Popis bezkontextové gramatiky pro vstupní program – 3 strany
II. Obecné popisy vybraných algoritmů
III. Třídní diagram
IV. Testovací programy
V. Aplikace IntProCan
VI. Zdrojové kódy aplikace

I. Popis bezkontextové gramatiky pro vstupní program

```
// ČÍSLA
digit -> 0 - 9

numberDecimal -> digit
               | numberDecimal digit

digitHexa -> 0 - 9 | a - f | A - F

numberHexa -> "0x" digitHexa
            | numberHexa digitHexa

number -> numberDecimal
        | numberHexa



// PROMĚNNÉ, NÁZVY
letter -> a - z | A - Z | "_"

character -> digit
           | letter

var -> letter
     | var character

labelname -> letter
           | labelname character

fncname -> letter
         | fncname character



// VÝRAZY, OPERACE
exprnumcomp -> number
             | expernumcomp "+" number
             | expernumcomp "-" number

exprnum -> number
         | "(" exprnumcomp ")"

unaryops -> var"++"
          | var"--"
          | "++"var
          | "--"var

expr -> var
      | number
      | fnccall
      | unaryops
      | exprnum "*" expr
      | expr "*" exprnum
```

```
      | expr "+" expr
      | expr "-" expr
      | "(" expr ")"

condition -> expr
           | condition "==" condition
           | condition "!=" condition
           | condition "<" condition
           | condition ">" condition
           | condition "<=" condition
           | condition ">=" condition

conditions -> condition
            | "!" conditions
            | conditions "&&" conditions
            | conditions "||" conditions
            | "(" conditions ")"



//PŘÍKAZY, BLOKY
command -> assignment ";"
         | fnccall ";"
         | goto ";"
         | return ";"
         | if
         | while
         | for
         | ";"

commandlabel -> command
              | label

commands -> empty
          | commandlabel
          | commands + commandlabel

block -> "{" commands "}"

statement -> command
           | block

assignment -> var "=" expr

parameters -> empty
            | variables

fnccall -> fncname "(" parameters ")"

goto -> "goto " labelname

label -> labelname ":"

return -> "return " expr
```

```
if -> "if" "(" conditions ")" statement
    | "if" "(" conditions ")" statement "else" statement

while -> "while" "(" conditions ")" statement

initfor -> empty
         | assignment

closefor -> empty
          | unaryops
          | assignment

for -> "for" "(" initfor ";" conditions ";" closefor ")" statement



// DEKLARACE PROMĚNNÝCH
variables -> var
           | variables "," var

decl -> "var" variables ";"

declarations -> empty
              | decl
              | declarations decl



// FUNKCE
fncbody -> "{" commands return ";" "}"

function -> "function" fncname "(" parameters ")" fncbody

functionmain -> "function" "main" "(" parameters ")" fncbody

functions -> empty
           | function
           | functions function



// PROGRAM
program -> empty
         | declarations functionmain functions
```

II. Obecné popisy vybraných algoritmů

II.1 Algoritmus přidání vektoru do množiny generátorů

Proměnné:
$\boldsymbol{i}$ = vedoucí index vektoru (obecně)
$\boldsymbol{i_G}$ = vedoucí index vektoru z $\boldsymbol{G}$
$\boldsymbol{i_x}$ = vedoucí index vektoru $\boldsymbol{x}$
$\boldsymbol{vp}$ = vedoucí prvek vektoru (obecně)
$\boldsymbol{vp_G}$ = vedoucí prvek vektoru z $\boldsymbol{G}$
$\boldsymbol{vp_x}$ = vedoucí prvek vektoru $\boldsymbol{x}$
$\boldsymbol{x_G}$ = vektor z $\boldsymbol{G}$

Vstup:
Generátor $\boldsymbol{G}$ - množina vektorů
Vektor $\boldsymbol{x}$ - vektor přidávaný do $\boldsymbol{G}$
Modulo $\boldsymbol{m} = 2^{\boldsymbol{w}}$

Algoritmus:

- Pokud neexistuje žádný $\boldsymbol{i_G} = \boldsymbol{i_x}$
  - Když $\boldsymbol{vp_x}$ je sudý
    - Vypočti $\boldsymbol{x_1} = 2^{\boldsymbol{w-r}} * \boldsymbol{x}$
    - pokud $\boldsymbol{x_1} \neq 0$
      - opakuj postup od začátku s $\boldsymbol{x_1}$ místo $\boldsymbol{x}$ (rekurzivně, hodnota $\boldsymbol{x}$ musí zůstat uložená)
  - přidej $\boldsymbol{x}$ do $\boldsymbol{G}$
  - seřaď vektory v $\boldsymbol{G}$ podle $\boldsymbol{i}$ od nejnižšího po nejvyšší
- pokud existuje $\boldsymbol{i_G} = \boldsymbol{i_x}$
  - vypočti rozklad
    - $\boldsymbol{vp_G} = \boldsymbol{d_G} * 2^{\boldsymbol{r_G}}$
    - $\boldsymbol{vp_x} = \boldsymbol{d_x} * 2^{\boldsymbol{r_x}}$
  - Když $\boldsymbol{r_G} > \boldsymbol{r_x}$
    - vyjmi $\boldsymbol{x_G}$ z $\boldsymbol{G}$
    - do $\boldsymbol{G}$ vlož $\boldsymbol{x}$ dle postupu od začátku
    - pokračuj s $\boldsymbol{x} = \boldsymbol{x_G}$ (tzn. považuj vektor v $\boldsymbol{G}$ za $\boldsymbol{x_G}$ a vkládej vektor $\boldsymbol{x}$, který má hodnoty dle původně vyjmutého $\boldsymbol{x_G}$)
  - Vypočti $\boldsymbol{x'} = \boldsymbol{d_G} * \boldsymbol{x} - \boldsymbol{2^{r_x - r_G}} * \boldsymbol{d_x} * \boldsymbol{x_G}$
  - Pokud $\boldsymbol{x'} \neq 0$
    - Nahraď $\boldsymbol{x} = \boldsymbol{x'}$
    - Opakuj postup od začátku

II.2 Algoritmus výpočtu matic zachycujících změny prováděné voláním funkcí

1. Vytvořit pracovní množinu (frontu) $W = \emptyset$. Fronta bude obsahovat dvojice (uzel, matice způsobující změnu).
2. Pro všechny uzly všech grafů nastavit množinu generující změnové matice pro daný vrchol na prázdnou množinu (množina generující změnové matice pro vrchol u bude dále v textu značena $S(u)$)
3. Pro všechny vstupní stavy všech funkcí:
   3.1. $S(st_q)$, tzn. množiny generující změnové matice pro vstupní stav funkce q, nastavit na jednotkovou matici rozměru (N+1) x (N+1), kde N je počet proměnných, (jednotková matice označována I_{N+1}).
   3.2. Do fronty ke zpracování vložit vstupní stav a jeho jednotkovou matici, tzn. $W = W \cup \{(st_q, I_{N+1})\}$
4. Dokud $W \neq \emptyset$:
   4.1. Vyjmi první dvojici (u, M) z W, kde u je prozkoumávaný uzel a M je matice, která způsobuje změnu, kterou je třeba propagovat dále
   4.2. Pokud je u výstupní uzel nějaké funkce q:
      4.2.1. Pro každou funkci, která funkci q volala (volání funkce q slouží k přechodu mezi uzly u' a v, které oba náležejí do nějaké funkce r):
         4.2.1.1. Vynásobit zleva maticí M každou matici z množiny řešení uzlu u', který funkci q volal. Tím vznikne množina $new = \{M * M_1 \mid M_1 \in S(u')\}$
         4.2.1.2. Pro všechny matice $M_2 \in new$:
            4.2.1.2.1. Převést M_2 na vektor a ten přidat do $S(v)$ pomocí algoritmu II.1
            4.2.1.2.2. Pokud došlo v přechozím kroku ke změně množiny $S(v)$
               4.2.1.2.2.1. Na konec W přidat nový nezpracovaný uzel, tj. $W = W \cup \{(v, M_2)\}$
   4.3. Pro všechny hrany s, které vychází z u do v:
      4.3.1. Vypočíst množinu new, která obsahuje výsledky násobení $M_1 * M$, kde M_1 je matice z množiny generující změnové matice pro přechod hranou s do uzlu v
      4.3.2. Pro všechny matice $M_2 \in new$:
         4.3.2.1. Převést M_2 na vektor a ten přidat do $S(v)$ pomocí algoritmu II.1
         4.3.2.2. Pokud došlo v předchozím kroku ke změně množiny $S(v)$
            4.3.2.2.1. Na konec W přidat nový nezpracovaný uzel, tj. $W = W \cup \{(v, M_2)\}$

II.3 Algoritmus výpočtu množin generátorů ve vrcholech CFG

1. Pro všechny vrcholy nastavit množinu generátorů všech řešení $R(u) = \emptyset$
2. Vstupní uzel programu je inicializován jednotkovými vektory
3. Vytvořit frontu požadavků ke zpracování z vektorů z bodu 2. W je seznam obsahující vždy dvojice (uzel, vektor), na počátku tedy bude $W = \{(u_0, e_0), \dots, (u_0, e_{N+1})\}$, kde N je počet proměnných a u_0 je vstupní uzel programu
4. Dokud $W \neq \emptyset$
   - 4.1. Vyjmi z W první dvojici (u, x), kde u je uzel a x je vektor, který způsobil změnu daného uzlu
   - 4.2. Pro všechny hrany s, které vedou z u do nějakého v:
     - 4.2.1. Pokud je s voláním nějaké funkce q, pak:
       - 4.2.1.1. Přidat x do množiny řešení vstupního uzlu funkce q (označen st_q) pomocí algoritmu II.1
       - 4.2.1.2. Pokud byla v předchozím kroku změněna množina $R(st_q)$, přidat do fronty nový uzel ke zpracování $W = W \cup (st_q, x)$
     - 4.2.2. Vypočíst množinu new, která obsahuje výsledky výpočtů $M * x$, kde M je matice z množiny matic pro přechod hranou s do uzlu v
     - 4.2.3. Pro všechny vektory $x' \in new$:
       - 4.2.3.1. Přidat x' do $R(v)$, tj. množiny řešení uzlu v, pomocí algoritmu II.1
       - 4.2.3.2. Pokud byla v předchozím kroku změněna množina $R(v)$, přidat do fronty nový uzel ke zpracování $W = W \cup \{(v, x')\}$

II.4 Algoritmus výpočtu diagonální matice

Proměnné:
$\boldsymbol{a_{ij}}$ – prvek matice $\boldsymbol{A}$ na $\boldsymbol{i}$-tém řádku v $\boldsymbol{j}$-tém sloupci
$\boldsymbol{x}, \boldsymbol{y}$ **–** prvky matice $\boldsymbol{A}$
$\boldsymbol{z}$ – prvek matice $\boldsymbol{T}$

Vstup:
Čtvercová $(N \times N)$ matice $\boldsymbol{A}$ – řešená soustava rovnic
Čtvercová $(N \times N)$ matice $\boldsymbol{T}$ – jednotková matice

Algoritmus – úprava $\boldsymbol{A}$ na diagonální matici:

- vypočti rozklad $\boldsymbol{a_{ij} = d_{ij} * 2^{r_{ij}}}$ pro všechna $\boldsymbol{a_{ij} \in A}$ a ulož hodnotu $\boldsymbol{i}$ a $\boldsymbol{j}$ pro $\boldsymbol{a_{ij}}$ s nejnižší hodnotou $\boldsymbol{r_{ij}}$ (pivotní člen)
- pro každý prvek $\boldsymbol{x \in A}$ ve sloupci $\boldsymbol{j}$, kromě pivotního členu, proveď:
  - vypočti $\boldsymbol{\alpha = x \gg r_{ij}}$
  - pro všechny prvky $\boldsymbol{y \in A}$, které jsou ve stejném řádku jako $\boldsymbol{x}$, proveď:
    - $\boldsymbol{y = y * d_{ij} - y_p * \alpha}$ (kde $\boldsymbol{y_p}$ je prvek z řádku $\boldsymbol{i}$ se stejným sloupcovým indexem, jako má $\boldsymbol{y}$)
- pro každý prvek $\boldsymbol{x \in A}$ v řádku $\boldsymbol{i}$, kromě pivotního členu, proveď:
  - vypočti $\boldsymbol{\alpha = x \gg r_{ij}}$
  - pro všechny prvky $\boldsymbol{y \in A}$ a $\boldsymbol{z \in T}$, které jsou ve stejném sloupci jako $\boldsymbol{x}$, proveď:
    - $\boldsymbol{y = y * d_{ij} - y_p * \alpha}$ (kde $\boldsymbol{y_p \in A}$ je prvek ze sloupce $\boldsymbol{j}$ se stejným řádkovým indexem, jako má $\boldsymbol{y}$)
    - $\boldsymbol{z = z * d_{ij} - z_p * \alpha}$ (kde $\boldsymbol{z_p \in T}$ je prvek ze sloupce $\boldsymbol{j}$ se stejným řádkovým indexem, jako má $\boldsymbol{z}$)
- vyměň sloupec $\boldsymbol{j}$ a první sloupec v nezpracované části matice $\boldsymbol{A}$
- vyměň sloupec $\boldsymbol{j}$ a první sloupec v nezpracované části matice $\boldsymbol{T}$
- vyměň řádek $\boldsymbol{i}$ a první řádek v nezpracované části matice $\boldsymbol{A}$
- opakuj postup od začátku pro zbývající část matice

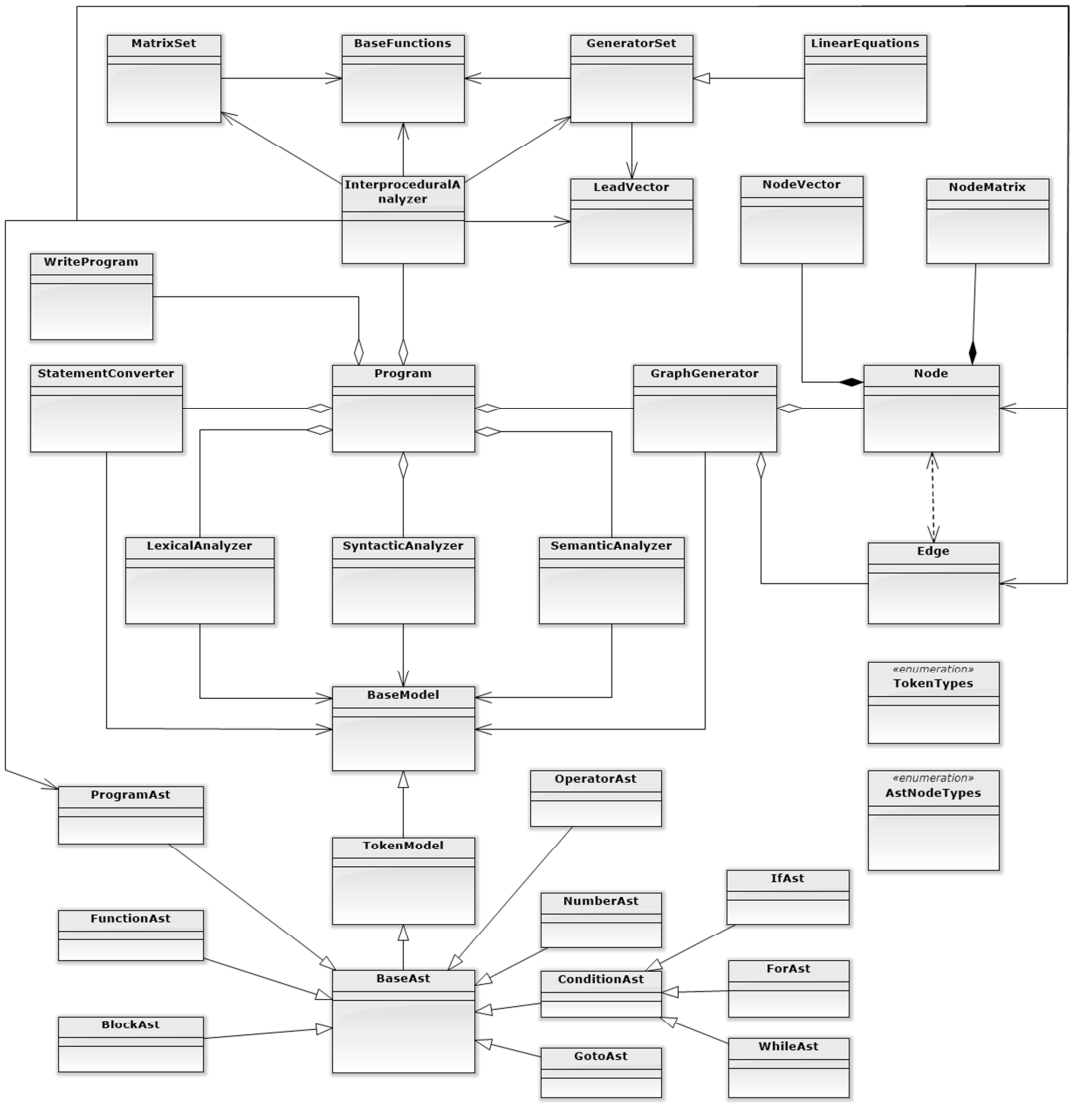
II.5 Algoritmus řešení homogenní soustavy lineárních rovnic z diagonální matice

Proměnné:
$\boldsymbol{a_{ij}}$ – prvek matice $\boldsymbol{D}$ na $\boldsymbol{i}$-tém řádku v $\boldsymbol{j}$-tém sloupci
$\boldsymbol{L_A}$ – množina všech řešení pro $\boldsymbol{Ax = 0}$
$\boldsymbol{G_{L_A}}$ – množina generátorů, generující $\boldsymbol{L_A}$
$\boldsymbol{l_i}$ – vektor z množiny generující modul řešení $\boldsymbol{Dx = 0}$
$\boldsymbol{e_i}$ – jednotkový sloupcový vektor s jedničkou v řádku $\boldsymbol{i}$
$\boldsymbol{x_i}$ – sloupcový vektor, $\boldsymbol{x_i} \in \boldsymbol{G_{L_A}}$
$\boldsymbol{w}$ – program je zpracováván v modulo 2^w

Vstup:
Čtvercová $(N \times N)$ matice $\boldsymbol{D}$ – diagonální matice, která vznikne z matice $\boldsymbol{A}$, a je výstupem algoritmu II.4
Čtvercová $(N \times N)$ matice $\boldsymbol{R^{-1}}$ – matice zachycující sloupcové úpravy provedené algoritmem II.4 na jednotkové matici $\boldsymbol{T}$, tato matice je výstupem algoritmu II.4

Algoritmus – výpočet $\boldsymbol{L_A}$
- pro každé $\boldsymbol{a_{ij}} \in \boldsymbol{D}$, kde $\boldsymbol{i = j}$ vypočti
  - $\boldsymbol{a_{ij} = d * 2^r}$
  - $\boldsymbol{l_i = 2^{w-r} * e_i}$, pro $\boldsymbol{2^r}$ vypočtené z $\boldsymbol{a_{ij}}$
  - $\boldsymbol{x_i = R^{-1} * l_i}$
  - Přidej $\boldsymbol{x_i}$ do $\boldsymbol{G_{L_A}}$ pomocí algoritmu II.1

III. Třídní diagram



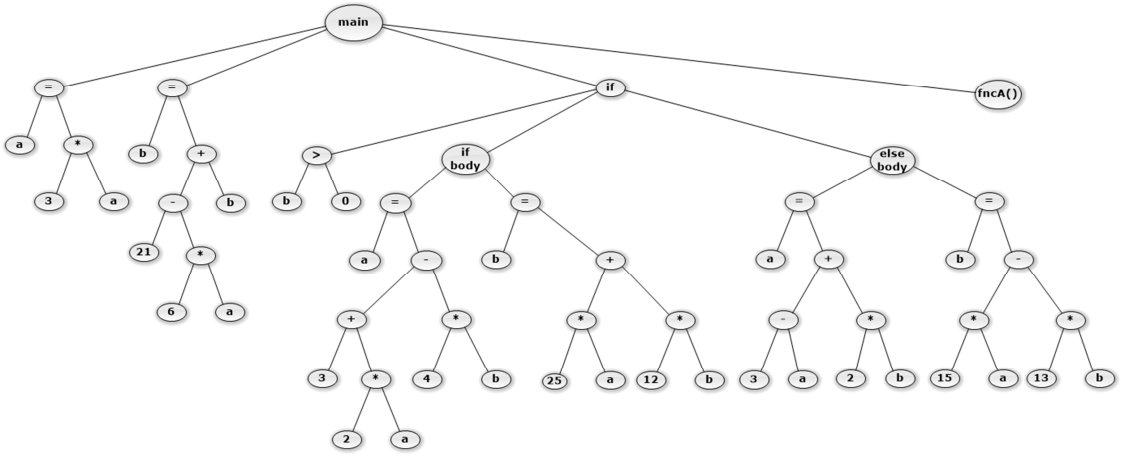
IV. Testovací programy

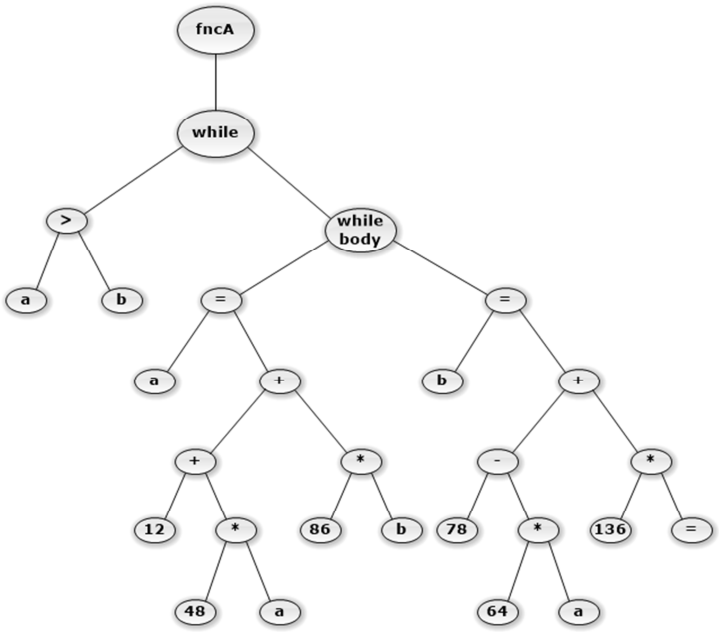
IV.1 Program 6.1.2

1) Kód:

```
var a,b;

function main()
{
  a = 3*a;
  b = 21 - 6*a + b;
  if ( b > 0 )
  {
    a = 3 + 2*a - 4*b;
    b = 25*a + 12*b;
  }
  else
  {
    a = 3 - a + 2*b;
    b = 15*a - 13*b;
  }

  fncA();
}

function fncA()
{
  while ( a > b )
  {
    a = 12 + 48*a + 86*b;
    b = 78 - 64*a + 136*b;
  }
}
```

2) AST:



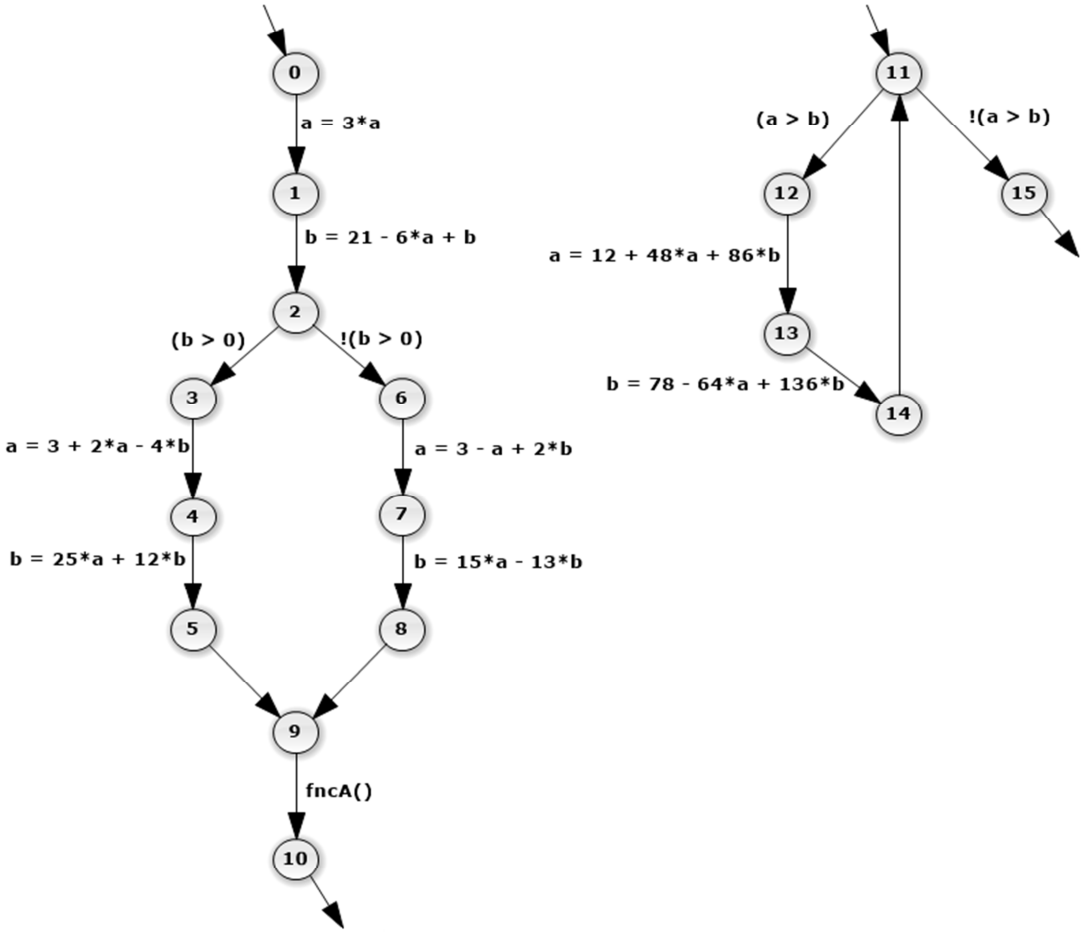


3) Posloupnost příkazů:

Manuální sestavení	Výstup aplikace

MAIN:
1: a = 3*a
2: b = 21 - 6*a + b
3: if !(b > 0) goto 7
4: a = 3 + 2*a - 4*b
5: b = 25*a + 12*b
6: goto 9
7: a = 3 - a + 2*b
8: b = 15*a - 13*b
9: fncA()
 	
FncA:
1: if !(a > b) goto 5
2: a = 12 + 48*a + 86*b
3: b = 78 - 64*a + 136*b
4: goto 1
5: end
	function main()
	{
	a = 3;
	b = ((21 - (6 * a)) + b);
	$IfBegin1:
	if (!((b > 0)))
	goto $IfFalse1;
	$IfTrue1:
	a = ((3 + (2 * a)) - (4 * b));
	b = ((25 * a) + (12 * b));
	goto $IfEnd1;
	$IfFalse1:
	a = ((3 - a) + (2 * b));
	b = ((15 * a) - (13 * b));
	$IfEnd1:
	fncA();
	}
	
	function fncA()
	{
	$WhileBegin2:
	if (!((a > b)))
	goto $WhileEnd2;
	$WhileTrue2:
	a = ((12 + (48 * a)) + (86 * b));
	b = ((78 - (64 * a)) + (136 * b));
	goto $WhileBegin2;
	$WhileEnd2:
	}

4) CFG:


0
a = 3*a
1
b = 21 - 6*a + b
2
(b > 0)
!(b > 0)
3
6
a = 3 + 2*a - 4*b
a = 3 - a + 2*b
4
7
b = 25*a + 12*b
b = 15*a - 13*b
5
8
9
fncA()
10
11
(a > b)
!(a > b)
12
15
a = 12 + 48*a + 86*b
13
b = 78 - 64*a + 136*b
14

IV.2 Program pro testování faktorů ovlivňujících rychlost zpracování

```
var a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t;

function main()
{
  a = 38 + 12*a - 44*b + 34*c - 82*d + 43*e - 252*f + 1868*g + 168*h +
26*i + 66*j + 38*k + 146*l - 164*m + 252*n - 246*o - 196*p - 459*q -
123*r - 46*s + 44*t;

  b = 42 + 38*a + 146*b - 164*c + 252*d - 246*e - 196*f - 459*g -
123*h - 46*i + 44*j + 12*k - 44*l + 34*m - 82*n + 43*o - 252*p +
1868*q + 168*r + 26*s + 66*t;

  c = 6*b + 86 - 222*c + 46*d - 198*e - 82*f + 24*g - 98*h + 38*i -
168*j - 2*k + 36*l + 246*m - 238*n + 26*o + 46*p - 254*q - 28*r - 42*s
- 68*t;

  d = 2*a + 36 + 246*c - 238*d + 26*e + 46*f - 254*g - 28*h - 42*i -
68*j + 6*k + 86*l - 222*m + 46*n - 198*o - 82*p + 24*q - 98*r + 38*s -
168*t;

  while ( b > 0 )
  {
    b = 34 + 16*a + 38*b + 28*c + 12*d + 9*e + 259*f + 369*g + 254*h +
28*i - 36*j + 84*k + 16*l + 12*m - 162*n + 186*o - 254*p + 968*q -
364*r + 68*s + 66*t;

    a = 64 + 24*a - 12*b + 46*c - 48*d + 194*e + 126*f + 164*g + 34*h
+ 86*i + 88*j - 36*k + 62*l - 86*m - 42*n - 246*o + 62*p - 186*q +
28*r - 28*s - 58*t;

    c = 22 - 14*a*b + 26*c + 238*d - 18*e - 248*f + 16*g - 68*h - 68*i
+ 44*j;

    d = 84*a + 16*b + 12*c - 162*d + 186*e - 254*f + 968*g - 364*h +
68*i + 66*j - 16*k + 38*l + 28*m + 12*n + 9*o + 259*p + 369*q + 254*r
+ 28*s - 36*t ;

    a++;
    b--;
    c++;
    d--;
  }
  fncA();
  b = 84*a + 16*b + 12*c - 162*d + 186*e - 254*f + 968*g - 364*h +
68*i + 66*j - 16*k + 38*l + 28*m + 12*n + 9*o + 259*p + 369*q + 254*r
+ 28*s - 36*t ;

  a = 2*a + 36 + 246*c - 238*d + 26*e + 46*f - 254*g - 34*h + 86*i +
88*j - 36*k + 62*l - 86*m - 42*n - 246*o + 62*p - 186*q + 28*r - 28*s
- 58*t;
}
```

```
function fncA()
{
  if (b*2 > a+c)
  {
    b = 42 + 126*a - 46*c - 38*d + 249*e - 166*f - 364*g - 46*h - 96*i
- 86*j + 32*k + 156*l + 82*m - 246*n + 38*o + 68*p - 498*q + 34*r +
38*s - 68*t;

    a = 38 + 12*a - 44*b + 34*c - 82*d + 43*e - 252*f + 1868*g + 168*h
+ 26*i + 66*j + 38*k + 146*l - 164*m + 252*n - 246*o - 196*p - 459*q -
123*r - 46*s + 44*t;

    fncB();
  }
  else
  {
    b = 22 - 14*a*b + 26*c + 238*d - 18*e - 248*f + 16*g - 68*h - 68*i
+ 44*j;
    fncC();
  }
  while(a-- > 0)
  {
    fncA();
    a--;
  }
  a = 2*a + 36 + 246*c - 238*d + 26*e + 46*f - 254*g - 34*h + 86*i +
88*j - 36*k + 62*l - 86*m - 42*n - 246*o + 62*p - 186*q + 28*r - 28*s
- 58*t;
}

function fncB()
{
  for (c = 1; c < a; c++)
  {
    a = 64 + 24*a - 12*b + 46*c - 48*d + 194*e + 126*f + 164*g + 34*h
+ 86*i + 88*j - 36*k + 62*l - 86*m - 42*n - 246*o + 62*p - 186*q +
28*r - 28*s - 58*t;

    b = 22 - 14*a*b + 26*c + 238*d - 18*e - 248*f + 16*g - 68*h - 68*i
+ 44*j;

    fncA();
  }

  a = 24 + 36*a + 62*b - 86*c - 42*d - 246*e + 62*f - 186*g + 28*h -
28*i - 58*j - 24*k - 12*l + 46*m - 48*n + 194*o + 126*p + 164*q + 34*r
+ 86*s + 88*t;

  if(b > 0)
  {
    fncD();
  }
}
```

```
function fncC()
{
  while (++a > b--)
  {
    a = 32*b + 156 + 82*c - 246*d + 38*e + 68*f - 498*g + 34*h + 38*i
- 68*j + 126*k - 46*l - 38*m + 249*n - 166*o - 364*p - 46*q - 96*r -
86*s;

    b = 42 + 126*a - 46*c - 38*d + 249*e - 166*f - 364*g - 46*h - 96*i
- 86*j + 32*k + 156*l + 82*m - 246*n + 38*o + 68*p - 498*q + 34*r +
38*s - 68*t;
  }

  a = 2*a + 36 + 246*c - 238*d + 26*e + 46*f - 254*g - 28*h - 42*i -
68*j + 6*k + 86*l - 222*m + 46*n - 198*o - 82*p + 24*q - 98*r + 38*s -
168*t;

  b = 6*b + 86 - 222*c + 46*d - 198*e - 82*f + 24*g - 98*h + 38*i -
168*j - 2*k + 36*l + 246*m - 238*n + 26*o + 46*p - 254*q - 28*r - 42*s
- 68*t;
}

function fncD()
{
  a = 24 + 36*a + 62*b - 86*c - 42*d - 246*e + 62*f - 186*g + 28*h -
28*i - 58*j - 24*k - 12*l + 46*m - 48*n + 194*o + 126*p + 164*q + 34*r
+ 86*s + 88*t;

  b = 84*a + 16*b + 12*c - 162*d + 186*e - 254*f + 968*g - 364*h +
68*i + 66*j - 16*k + 38*l + 28*m + 12*n + 9*o + 259*p + 369*q + 254*r
+ 28*s - 36*t ;

  fncD();

  a = 38 + 12*a - 44*b + 34*c - 82*d + 43*e - 252*f + 1868*g + 168*h +
26*i + 66*j + 38*k + 146*l - 164*m + 252*n - 246*o - 196*p - 459*q -
123*r - 46*s + 44*t;
}
```

IV.3 Krátký program pro testování faktorů ovlivňujících rychlost zpracování algoritmů

```
var a = 2;
var b = 3;
var c = 4;
var d = 5;
var e = 6;
var f = 7;
var g = 8;
var h = 9;
var i = 10;
var j = 11;
var k = 12;
var l = 13;
var m = 14;
var n = 15;
var o = 16;
var p = 17;
var q = 18;
var r = 19;
var s = 20;
var t = 21;

function main()
{
  a = b + c + d + e + f + g + h + i + j + k + l + m + n + o + p + q +
r + s + t;
  fncA();
}

function fncA()
{
  b = 6 + 3*a + 4*c + 6*d - 8*e + 9*f + 8*g + 50*h + 46*i + 70*j +
46*k + 70*l + 15*m + 46*n + 72*o + 92*p + 84*q + 6*r + 4*s + 2*t;
  a = 3*b + c + 2*d + 7*e + 6*f + 5*g + 79*h + 86*i + 42*j + 46*k +
82*l + 46*m + 82*n + 42*o + 78*p + 16*q + 24*r + 92*s + 54*t;
}
```